VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Seznámení se s technologií Composite WPF (Prism) a ověření technologie v ukázkové aplikaci

# Introducing the Composite WPF (Prism) Technology and its Verification in a Sample Application

2010 Bc. Jan Golasowski

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 22. dubna 2010 .....

Především bych rád poděkoval svým rodičům, kteří mne podporovali po celou dobu mého studia. Pomáhali mi pokaždé, když jsem to potřeboval a ve všech situacích mi byli oporou.

Dále bych rád poděkoval vedoucímu mé práce, Ing. Tomáši Kocyanovi, za jeho vstřícnost a ochotu, se kterou mi pomáhal řešit vzniklé problémy.

## Abstrakt

Technologie Composite WPF, označovaná jako Prism V2, slouží pro vývoj kompozitních aplikací pro Windows Presentation Foundation. Jejím cílem je usnadnit vývoj těchto aplikací a to zejména s pomocí návrhových vzorů typu Inversion of Control a Separated Presentation. Samotný balík knihoven, kterým Prism v2 je, však mnohdy pro vývoj rozsáhlých kompozitních aplikací nestačí. Jedná se především o problém použití vzoru Model-View-ViewModel pro zobrazování modálních a nemodální oken, jejich následné zavírání apod. Tato práce slouží jednak pro seznámení s technologií Prism, ale zabývá se i otázkou řešení výše jmenovaných problémů.

**Klíčová slova:** kompozitní aplikace, moduly, Prism, CAL, návrhové vzory, Model-View-ViewModel

## Abstract

The Composite WPF technology, known as Prism V2, is a set of libraries that helps with development of composite applications for Windows Presentation Foundation. The main goal of Prism is to simplify development of composite applications with utilisation of design patterns. Especially the Inversion of Control and Separeted Presentation design patterns. There are many problems in the world of huge and complicated composite applications. This type of application usually uses modal and non-modal windows for example. It is not easy to apply Model-View-ViewModel design pattern on modal window and it is complicated to close that window from ViewModel. This thesis describes the Prism itself and tries to solve the problems mentioned above.

**Keywords:** composite application, modules, Prism, CAL, design patterns, Model-View-ViewModel

## Seznam použitých zkratek a symbolů

| | | |
|---|---|---|
| CAL | – | Composite Application Library |
| DI | – | Dependency Injection |
| IoC | – | Inversion of Control |
| LOB | – | Line-of-business |
| PM | – | Presentation Model |
| RIA | – | Rich Internet Application |
| SL | – | Service Locator |
| SC | – | Supervising Controller |
| TDI | – | Tabbed document interface |
| TDD | – | Test-driven Development |
| MDI | – | Multiple document interface |
| MVC | – | Model-View-Controller |
| MVP | – | Model-View-Presenter |
| MVVM | – | Model-View-ViewModel |
| UI | – | User Interface |
| URI | – | Uniform Resource Identifier |
| VM | – | ViewModel |
| XAML | – | Extensible Application Markup Language |
| WPF | – | Windows Presentation Foundation |

# Obsah

# Seznam obrázků

# Seznam výpisů zdrojového kódu

# 1 Úvod

V dnešní době se musí vývojáři aplikací vypořádat s mnoha problémy. Během vývoje dochází často k situacím, kdy zákazník častokrát mění požadavky na výslednou aplikaci, objevují se nové technologie a postupy, které mění dosavadní styl práce. Proto je důležité vytvářet aplikace tak, aby bylo možné je během vývoje snadno upravovat a také rozšiřovat jejich funkčnost. Takováto aplikace nemůže být vytvořena jako jednotný celek, kdy jednotlivé komponenty jsou spolu těsně svázány a není mezi nimi jasně definována hranice. Do takto navržené aplikace se pak velmi obtížně přidávají nové funkce nebo upravují ty stávající. Je obtížně aplikaci testovat a oprava jedné chyby často způsobí vznik chyby v jiné části aplikace.

Řešením těchto problémů je rozdělit aplikaci do jednotlivých bloků, tzv. modulů, kdy každý z nich má na starosti řešení nějaké specifické funkční oblasti. Tyto bloky je možné vyvíjet nezávisle na ostatních, můžeme je velmi jednoduše testovat a také můžeme práci na projektu rozdělit mezi více týmů, čímž se zvýší efektivita práce. Aplikacím vytvořeným pomocí tohoto principu, tedy pomocí skládání modulů do jednoho celku, se říká *kompozitní*.

## 1.1 Kompozitní aplikace

Kompozitní aplikací nazýváme takovou aplikaci, která je složená z několika diskrétních, slabě vázaných (tím máme na mysli, že jedna komponenta není přímo závislá na druhé) komponent, které jsou začleněny do tzv. *shellu*. Shell většinou chápeme jako jedno hlavní okno celé aplikace, do kterého jsou jednotlivé komponenty a moduly vloženy.

Kompozitní aplikace mají mnoho výhod, jsou to například tyto:

- Moduly mohou být vyvíjeny, testovány a předávány do provozu nezávisle na sobě. Mohou být snadno upravovány a rozšiřovány, což usnadňuje rozšiřování celé aplikace a její následnou správu.
- Moduly jsou integrovány do jednoho shellu, kde spolu komunikují pomocí slabé vazby. Shell zajistí jednotný vzhled celé aplikace a přístup k funkcím modulů pomocí jednotlivých komponent uživatelského rozhraní.
- Moduly jsou od sebe jasně odděleny a je možné je pak znovu použít při vývoji další aplikace.
- Rozdělení do modulů umožní také rozdělit práci na projektu mezi více týmů, kdy jeden tým může pracovat na uživatelském rozhraní a druhý na vývoji aplikační logiky.

Ovšem navrhnout a vytvořit dobře fungující kompozitní aplikaci není snadné. Velkým problémem je například ono zajištění slabé vazby mezi jednotlivými moduly. Proto vzniklo a stále vzniká mnoho různých nástrojů a knihoven, které vývoj kompozitních aplikací usnadňují. Mezi ně také patří *Composite Application Library*.

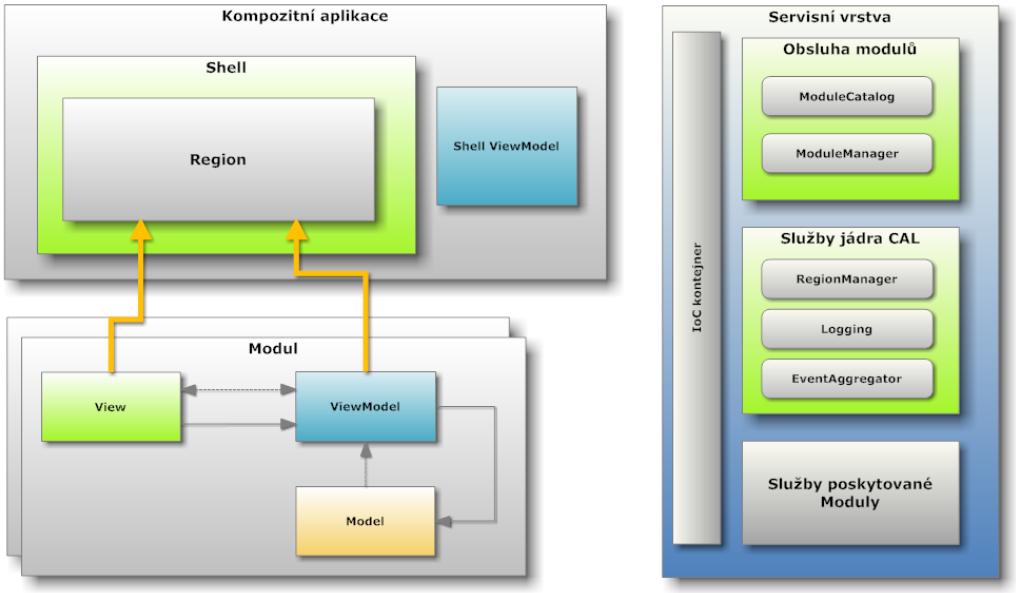
## 1.2 Composite Application Library

Composite Application Library (dále jen CAL) je sada knihoven, které usnadňují vývoj kompozitních aplikací, a to konkrétně pro technologie Windows Presentation Foundation (WPF) a Silverlight. CAL urychluje vývoj kompozitních aplikací použitím ověřených návrhových vzorů (viz 2 na straně 9), které slouží k tvorbě takovéhoto typu aplikací.

Než se však dostaneme k popisu jednotlivých návrhových vzorů, které jsou v aplikacích tvořených pomocí CAL používány, popíšeme si, jak vypadá architektura samotné CAL.

## 1.3 Základní architektura CAL

Architektura CAL (viz Obrázek 1) se skládá z aplikace, modulů a servisní vrstvy. Aplikace se skládá z shellu, který definuje jednotlivé regiony, do kterých jsou následně vloženy jednotlivé moduly. Moduly obsahují tzv. view a služby, které jsou pro modul specifické. Pod úrovní aplikace pak leží servisní vrstva, která poskytuje přístup k různým službám modulů a CAL.




Obrázek 1: Základní architektura CAL

Jednotlivé části architektury kompozitní aplikace jsou následující:

- **Shell** - Jedná se o hlavní okno celé aplikace, které poskytuje místo pro konkrétní prvky uživatelského rozhraní jednotlivých modulů. Shell definuje rozvržení celé aplikace a většinou nemá ponětí o tom, jaké komponenty bude obsahovat. Sám o sobě obsahuje minimální funkčnost, ta leží spíše v jednotlivých modulech.

- **Shell presenter** - veškerá prezentační logika shellu patří sem. Jedná se o konkrétní aplikaci jednoho z návrhových vzorů Separated presentation (viz 3 na straně 19) , čím se oddělí vzhled uživatelského rozhraní od prezentační logiky.
- **Regiony** - jedná se o prvky rezervující místo v shellu, do kterého budou umístěny prvky uživatelského rozhraní jednotlivých modulů. Moduly mohou k regionům přistupovat pomocí RegionManageru a poté do vybraného regionu vložit svůj obsah.
- **Moduly** - množina modulů, tedy view a služeb, které spolu logicky spolupracují, ale mohou být vyvíjeny a testovány nezávisle na sobě. V kompozitní aplikaci se moduly nejprve vyhledají a poté načtou. CAL používá následující postup:
  1. *Naplnění tzv. Module catalog* (katalog modulů obsahující seznam jednotlivých modulů spolu s jejich metadaty). Katalog můžeme naplnit několika způsoby - pomocí kódu, deklarativně pomocí XAML, pomocí konfiguračního souboru a také můžeme načíst všechny moduly, které se nacházejí v určité složce na disku.
  2. *Načtení modulů* - jedná se o uložení modulu na lokální úložiště, pokud se moduly např. stahují z internetu, a načtení tohoto modulu do aplikace. Moduly se mohou načítat automaticky nebo jsou načítány až po explicitním vyžádání, tzv. OnDemand.
  3. *Inicializace modulů* - moduly jsou inicializovány hned, jakmile jsou dostupné.
- **View** - view mají na starosti zobrazování obsahu modulů na obrazovku. Jedná se tedy o skupinu ovládacích prvků, které umožňují přístup k funkcím modulu. V kompozitní aplikaci jsou view často složeny z dalších view. View by mělo být testovatelné a používat odpovídající návrhové vzory pro Separated Presentation (viz 3 na straně 19). Měla by se také používat funkce databindingu kdekoliv je to možné.
- **Komunikace** - jednotlivé komponenty aplikace mají většinou potřebu komunikovat s ostatními komponentami, ať už jsou ze stejného nebo jiného modulu. CAL poskytuje dvě možnosti řešení. Jsou to CompositeCommand a EventAggregator.
  - **CompositeCommand** - když je aplikace složena z několika view, je často také nutné sloučit příkazy, které jednotlivá view podporují, do jednoho. CompositeCommand představuje právě takovéto sloučení, které umožní tomu, kdo CompositeCommand vyvolal, ovlivnit více příkazů najednou.
  - **EventAggregator** - view, která potřebují poslat nějakou událost ostatním view nebo komponentám, ale nevyžadují jejich odpověď, používají právě EventAggregator. Libovolný počet komponent může s jeho pomocí rozesílat události a stejně tak si může libovolné mnoho komponent vyžádat příjem konkrétní události.
- **Services** - služby, které aplikace a jednotlivé moduly nabízejí, ať už pro vlastní nebo cizí využití. Služby jsou nabízeny pomocí tzv. kontejneru, který služby lokalizuje a

častokrát také vytváří. CAL používá pro lokalizaci těchto služeb (tedy jako Service Locator, viz 2.3 na straně 15) kontejner Unity.

K detailnímu popisu jednotlivých částí CAL se dostaneme v části 4 na straně 29. Nyní se budeme věnovat zmiňovaným návrhovým vzorům, které se v aplikacích vyvíjených pomocí CAL velmi často používají a prakticky se bez nich neobejdeme.

# 2 Návrhové vzory v Composite Application Library

Návrhové vzory, které je nutné znát proto, abychom byli schopni vyvíjet kompozitní aplikace pomocí CAL, můžeme rozdělit do dvou skupin. Jedná se o:

1. Skupinu vzorů, které se zabývají řešením problémů zavislých komponent.
2. Návrhové vzory, které pomáhají oddělit vzhled uživatelského rozhraní od implementace aplikační logiky.

Do první skupiny, která řeší problém známý pod názvem *Inversion of Control*, patří vzory *Dependency Injection* a *Service Location*. Do druhé skupiny, která je označována jako skupina vzorů *Separated Presentation*, pak patří *Model-View-Controller*, *Model-View-Presenter* a hlavně *Model-View-ViewModel*. Všechny tyto vzory si v následujících částech popíšeme.

## 2.1 Inversion of Control (IoC)

Inversion of Control, nebo také obrácené řízení, je návrhový vzor, který umožňuje uvolnit vztahy mezi jinak těsně svázanými komponentami.

### Popis problému

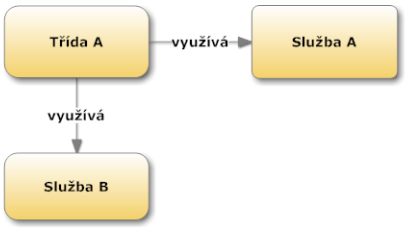
Máme třídu, která je závislá na dalších komponentách (viz Obrázek 2). Aby mohla třída tyto závislosti používat, musí být konkrétní implementace těchto závislostí známá již v době kompilace. Chceme-li změnit závislosti nebo aktualizovat jejich kód, znamená to nutnost provádět změny i v naší třídě. Třídu je obtížné testovat izolovaně, protože má přímé reference na své závislosti, které musí být pro otestování funkční. To znamená, že pro testování třídy nemůžeme použít tzv. Mock [1] a Stub implementace závislosti, což testování značně komplikuje a prodlužuje.

### Použití IoC

Hlavním důvodem proč použít tento vzor je situace, kdy chceme oddělit třídu od jejích závislostí. Snažíme se tedy vyhnout tomu, abychom v těle třídy ručně vytvářeli instance závislých komponent a služeb, se kterými chceme v naší třídě pracovat.

Pokud totiž vytváříme instance závislostí v těle třídy ručně, vytvoříme tak mezi naší třídou a jejími závislostmi silnou vazbu. Pokud pak dojde k situaci, kdy musíme udělat změny v implementaci některé ze závislostí, bude to pro nás velmi často znamenat i nutnost upravit implementaci naší třídy. Jednoduchým příkladem je např. nový parametr v

[1] Mock implementace objektu slouží pro provádění automatických testů. V podstatě se jedná o nahrazení reálného objektu testovací fasádou, která neprovádí žádnou funkcionalitu nahrazovaného objektu, ale jen se jako tento objekt tváří. Místo původní logiky objektu je vloženo chování, které v našem testu potřebujeme.
Klasickým příkladem může být simulace načítání dat do kolekce datových modelů z databáze. Zde místo skutečného výsledku získaného z databáze použijeme staticky naplněnou kolekci, kterou vytvoříme přímo v kódu.




Obrázek 2: Inversion of Control

konstruktoru závislosti. Pokud se totiž během tvorby aplikace ukáže, že potřebujeme v závislé komponentě pracovat s dalším objektem, který bude závislé komponentě předám jako parametr konstruktoru, musíme upravit také implementaci naší třídy. Jakmile instanci závislosti vytváříme ručně, musíme se totiž postarat o zajištění všech parametrů, které vstupují do konstruktoru této závislosti. V tomto případě to znamená zajistit získání instance objektu, která bude vytvářené instanci závislosti předána. Situace se tímto značně komplikuje.

Aplikujeme-li však vzor IoC, velmi si tuto situaci usnadníme. O vytvoření a předání instance objektu, který je použit jako nový parametr v konstruktoru závislosti, se totiž musí postarat IoC kontejner. Nemusíme tak provádět v implementaci naší třídy prakticky žádné změny. Tento vzor nám také usnadňuje použití závislostí, jejichž konkrétní implementace není známá v době kompilace, ale bude známa až v průběhu životního cyklu aplikace. Může se například jednat o komponenty, které budou staženy z webu apod. V naší třídě se tak nemusíme zabývat ani tím, odkud závislou komponentu získáme, protože vše má na starosti IoC kontejner. Bude sice nutné provézt určitá nastavení kontejneru, ale do implementace naší třídy, nebudeme muset zasahovat.

Další velkou výhodou použití IoC je ulehčení a urychlení testování. Můžeme totiž použít zmiňované Mock a Stub implementace závislostí, které jednak testování urychlí a jednak odstíní chyby závislostí od chyb v testované třídě. K této problematice se dostaneme v části 6 na straně 60.

### Řešení

Přenechat funkci vyhledání a výběru konkrétní implementace závislosti na externí komponentě či zdroji. V podstatě to znamená, že třída nevytváří sama instance dalších tříd, které potřebuje, ale jsou jí dodány nějakým způsobem z vnějšku.

### Implementace IoC

Návrhový vzor Inversion of Control může být implementován různými způsoby. Composite Application Library (dále jen CAL) používá dva typy implementace a jsou to *Dependency Injection* a *Service Locator*. Jak se od sebe jednotlivé implementace liší a způsob jejich použití si ukážeme v následujících částech.

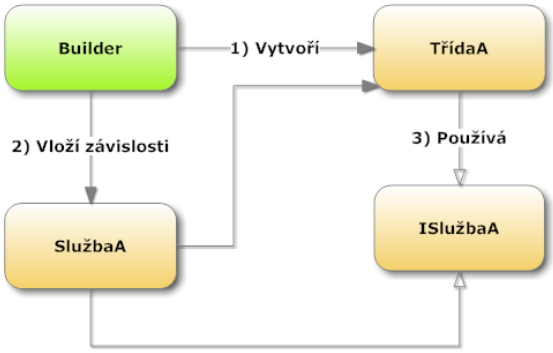
### Nevýhody IoC

Stejně jako každý návrhový vzor má kromě svých výhod také nějaké nevýhody, tak i použití IoC má svá negativa. Pokud chceme v naší aplikaci vzor IoC použít, musíme zajistit mechanizmus, který nám v průběhu inicializace třídy poskytne veškeré potřebné závislosti. Nejčastěji tak použijeme jednu z existujících implementací některého z IoC kontejnerů, který bude potřebný mechanizmus realizovat. Vytvoření vlastního IoC kontejneru je sice možné, ale jedná se o zbytečnou ztrátu času. Existující implementace jsou totiž vývojáři dostatečně prověřeny a navíc je možné tyto implementace snadno rozšířit a upravit tak, aby se kontejner choval podle našich potřeb.

Další nevýhodou, která plyne z použití IoC, je skutečnost, že orientace v kódu, který tento návrhový vzor využívá, může být pro programátora obtížnější. Zvláště pak v případě, že daný programátor tento návrhový vzor nezná.

## 2.2 Dependency Injection (DI)

DI je v podstatě novější název pro IoC, jedná se už o konkrétní techniku využití IoC.




Obrázek 3: Dependency Injection

### Princip DI

Principem DI je nevytvářet instance jednotlivých závislých komponent přímo ve třídě, ale deklarativně tyto závislosti vyjádřit v její definici. Tyto závislé komponenty jsou obvykle reprezentovány spíše jako rozhraní než konkrétní implementace tříd. To usnadňuje testování třídy, neboť můžeme konkrétní implementace závislotí nahradit jejich Mock verzemi. Pro získání korektní instance komponenty, na které je třída závislá, používáme objektu Builder (viz Obrázek 3), což je jakýsi kontejner, který požadovanou instanci třídě dodá během jejího vytváření a inicializace.

Existuje několik typů DI, viz [1]:

- **Constructor Injection** - používáme parametry konstruktoru třídy k vyjádření jejích závislostí a také k získání instancí těchto závislostí v průběhu vytváření třídy. Jako objekt Builder pro získávání instancí těchto závislostí a jejích vložení do třídy může sloužit například *UnityContainer*.
- **Setter Injection** - závislosti jsou vyjádřeny pomocí setter metod, které používá objekt Builder pro získání instancí těchto závislostí a jejich vložení do třídy během její inicializace.
- **Interface Injection** - nejprve je definováno rozhraní, které určuje metody, pomocí nichž budou nastavovány instance objektů, které příslušná třída potřebuje. Každá třída, která chce tyto instance využívat, musí implementovat příslušné rozhraní.

Dnešní frameworky, stejně tak CAL, používají převážně Constructor a Setter Injection, a to z toho důvodu, že v případě Interface Injection se musí vytvořit mnoho rozhraní pro získání všech závislostí.

V případě *Constructor Injection* a Setter Injection není potřeba ve třídě provádět žádné speciální úpravy a stejně tak i kontejner provádějící vkládání závislostí nemá složitou práci při získávání instancí těchto závislostí. Bývá výhodou poskytnout obě metody pro vkládání závislostí. Constructor Injection má však výhody v tom, že ihned vidíme, jaké závislosti třída ke své činnosti potřebuje a navíc je jeho získání umístěno na správném místě, tedy tam, kde třída vzniká. Pokud je možné třídu vytvořit více cestami, je vhodné napsat také více konstruktorů, které tvorbu zajistí. Další výhodou Constructor Injection je možnost skrýt neměnné vlastnosti třídy jednoduše tak, že k nim nevytvoříme Setter.

V případě použití *Setter Injection* se toto stává problémem, neboť pro první získání instance závislosti použijeme Setter metodu, ale pak již nechceme umožnit tuto závislost změnit. Setter metoda je však stále přístupná, což nebrání ve změně instance. Výhoda Setter Injection spočívá naopak v tom, že pokud potřebujeme pro vytvoření třídy jednoduché parametry, např. string, tak můžeme setteru dát jméno, ze kterého je účel parametru ihned jasný. V případě konstruktoru jsme odkázáni na pozici tohoto parametru, což může být méně přehledné.

### Výhody DI

Tím, že použijeme vzor DI ve své aplikaci, si přináší mnoho výhod. Jedná se především o jednodušší tvorbu, úpravy a údržbu jednotlivých komponent aplikace, protože je mezi nimi slabá vazba. Výrazně jednodušší je také testování kódu, které spolu s volnějšími závislostmi patří k hlavním výhodám IoC obecně.

Použitím DI také odpadá povinnost zajišťovat korektní přetypování tříd, protože se o ně postará sám IoC kontejner. Pokud navíc použijeme Constructor Injection jsou závislé komponenty, se kterými budeme v naší třídě pracovat, vyjádřeny explicitně a velmi snadno je nalezneme. IoC kontejner ve většině případů nevyžaduje speciální zásah do kódu aplikace, jednotlivé komponenty je pak možné snadno využít i jinde, bez nutnosti přepisování kódu.

### Nevýhody

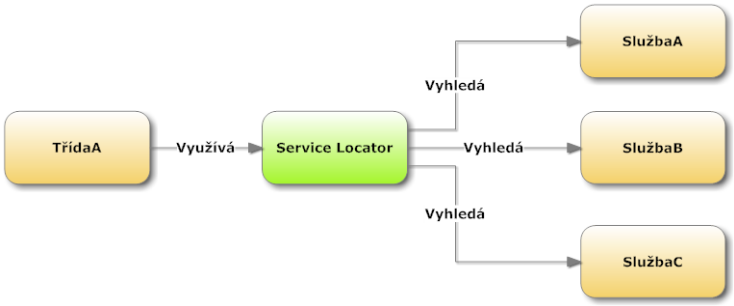
První nevýhodou, která přichází v souvislosti s použitím DI, je jistě zajištění toho, že IoC kontejner potřebuje znát všechny závislosti. Před vytvořením instance objektu, který je vytvořen pomocí DI totiž musí kontejner umět nalézt všechny implementace závislosti, které jsou pro vytvoření nového objektu potřebné. To pro programátora znamená nutnost, říci kontejneru, jaké konkrétní implementace odpovídají abstraktnějším rozhraním, která jsou použita v definici parametrů konstruktoru objektu, má kontejner použít. Kontejner je také zodpovědný za životní cyklus závislostí. Musíme proto navíc definovat, které závislosti budou vytvořeny pouze jednou a poté bude pro každé další použití vrácena reference na jejich instanci, tedy tzv. singletony a u kterých závislostí bude naopak pokaždé vytvořena nová instance.

Použitím DI si sice zajistíme slabou vazbu mezi naší třídou a závislostmi, ale také se kód stane složitějším pro správu. Musíme totiž v rámci projektu pracovat s více soubory

se zdrojovými kódy, než v případě, že DI nepoužijeme. Projekt totiž musí obsahovat soubory s implementací IoC kontejneru a většinou také soubory s definicí používaných rozhraní pro závislosti. Pro jednodušší aplikace je tak možná zbytečné zabývat se složitým kódem, ale je snazší napsat aplikaci klasickým způsobem.

## 2.3 Service Locator (SL)

Jedná se opět o konkrétní techniku využití IoC.




Obrázek 4: Service Locator

### Princip SL

Použijeme objekt SL, který obsahuje reference na služby, na nichž je naše třída závislá a který poskytuje možnost jejich nalezení (viz Obrázek 4). V naší třídě pak tento objekt použijeme pro získání instancí závislých služeb (komponent). Objekt SL většinou nevytváří instance závislých komponent (UnityContainer patří k výjimkám, viz Implementace SL), ale poskytuje způsob, jak tyto komponenty registrovat a poté je také nalézt. SL by měl poskytovat možnost pro nalezení závislé komponenty bez specifikace jejího konkrétního typu. Měl by například umožnit nalezení komponenty pomocí zadání jejího rozhraní.

### Implementace SL

CAL používá jako SL Unity kontejner, který umí vytvořit a získat instanci i neregistrované komponenty. Takovouto instanci získáme pomocí příkazu:

```
ServiceLocator.Current.GetInstance<T>(),
```

kde T je konkrétní typ instance, nesmí se jednat o rozhraní.

### Výhody SL

Výhody plynoucí z použití vzoru SL v naší aplikaci jsou stejné jako v případě použití vzoru DI, protože oba vzory jsou zobecněním návrhového vzoru IoC. Jedná se tedy především o zajištění slabé vazby mezi závislými komponentami a objekty, které tyto komponenty používají. Další velkou výhodou je opět snadnější a rychlejší testování. Navíc odpadá, v případě použití SL oproti DI, nutnost upravovat konstruktor třídy tak, aby jako parametry obsahoval potřebné závislosti.

### Nevýhody SL

K nevýhodám opět patří větší počet souborů se zdrojovými kódy, které musíme v rámci projektu spravovat. Navíc jsou zde oproti DI další nevýhody. Než budeme moci SL použít pro získání instance určité závislosti, musíme nejdříve napsat kód, který tuto závislost do objektu SL zaregistruje. Teprve pak bude SL vědět, kde má instanci potřebné závislosti získat.

Aplikace vzoru SL je sice výhodná v tom, že nemusíme upravovat konstruktor třídy, která používá závislé komponenty, ale je zde problém s přehledností. Získání instance závislosti pomocí objektu SL může být totiž použito v libovolném místě unitř těla třídy. Navíc se pak pro danou třídu stane objekt SL další závislou komponentou, což může být v některých situacích problematické. Tuto nevýhodu si blíže popíšeme v následující části.

## 2.4 Service Locator vs Dependency Injection

Obě metody slouží pro oddělení závislosti třídy na konkrétní implementaci komponenty, kterou využívá. Rozdíl je však v tom, jak je konkrétní implementace dané třídě poskytnuta.

V případě SL si o implementaci třída explicitně požádá, ale v případě DI zde žádná žádost není a implementace je přímo vložena do třídy. Použijeme-li SL, je jasné, že naše třída je na něm závislá. SL sice skryje závislost na konkrétních implementacích, ale pořád je zde závislost na samotném SL. Pokud je tato závislost problémem, je vhodné použít DI. DI má oproti SL tu výhodu, že je snadné najít závislosti dané třídy. Jsou uvedeny v konstruktoru. V případě SL je nutné procházet zdrojový kód a hledat místa, kde probíhá volání SL. Moderní IDE sice toto hledání usnadňují pomocí schopnosti nalezení reference, ale stále je to složitější než pohled na konstruktor.

Použití SL je výhodné v případě, že píšeme třídu, která bude využívat další různé třídy pro přístup k určitým zdrojům, kdy tyto třídy píší jiní vývojáři. Zde pak stačí, aby ostatní vývojáři nakonfigurovali náš SL tak, aby našel tu jejich implementaci. To mohou provézt například v konfiguračním souboru nebo přímo v kódu. Horší je situace v opačném případě. Tedy pokud píšeme třídu, která nevyužívá pro svou činnost jiné třídy, ale je sama používána jinými třídami. Problém spočívá v tom, že každá třída, která bude chtít naší třídu používat pomocí SL, může mít implementaci SL naprosto odlišnou od ostatních tříd. Dá se sice vytvořit k naší třídě rozhraní, ke kterému pak ostatní třídy poskytnou adaptér, s jehož pomocí budeme k jejich SL přistupovat, ale tím veškeré výhody přímého propojení naší třídy s SL přijdou vniveč.

## 2.5 Příklad použití vzorů DI a SL v CAL za pomoci IoC kontejneru

### Příklad DI - typ Constructor

```
1  private IMovieService Service;
2  private IEventAggregator EventAggregator;
3
4  public MovieViewModel(IMovieService service, IEventAggregator aggregator)
5  {
6      EventAggregator = aggregator;
7      Service = service;
8      Movies = new BindableCollection<MoviePresentationModel>();
9  }
10
11 private void SortMovies(object notUsed)
12 {
13     var temp = Movies.OrderByDescending(m => m.Price).ToList();
14     foreach (MoviePresentationModel model in temp) {    Movies.Add(model); }
15     EventAggregator.GetEvent<EventMoviesSorted>().Publish(String.Empty);
16 }
```

Výpis 1: Zdrojový kód třídy MovieViewModel

Na řádku číslo 4 v předchozím výpisu vidíme zápis konstruktoru třídy MovieViewModel. Ten má jako své závislosti jednak service, který načíta data z internetu, a jednak také aggregator, které ma na starosti předávání událostí mezi moduly.

### Příklad SL

```
1  private IMovieService Service;
2  private IEventAggregator EventAggregator;
3
4  public MovieViewModel()
5  {
6      EventAggregator = ServiceLocator.Current.GetInstance<IEventAggregator>();
7      Service = ServiceLocator.Current.GetInstance<IMovieService>();
8      Movies = new BindableCollection<MoviePresentationModel>();
9  }
```

Výpis 2: Zdrojový kód třídy MovieViewModel

V tomto příkladu vidíme opět konstruktor třídy MovieViewModel, tentokrát však jeho druhou variantu - použití SL. Pro získání závislostí zde používáme statickou třídu ServiceLocator, která nám vrátí instanci požadované závislosti. Jak vidíme, o závislost je nutné explicitně požádat.

V obou případech však musíme nejprve závislosti[2], které budeme později používat, registrovat v našem kontejneru, který slouží jako Locator v případě SL nebo jako objekt Builder v případě DI. Příklad registrace uvádí výpis 3

### Příklad registrace závislosti

```
1  public MovieModule(IUnityContainer container, IRegionManager regionManager)
2   {
3      this.container = container;
4     this.regionManager = regionManager;
5      Container.RegisterInstance<IMovieService>(new MovieService());
6  }
```

Výpis 3: Zdrojový kód třídy MovieModule

[2]Kromě objektu IEventAggregator, použitého v příkladech, který je součástí servisní vrstvy CAL a tudíž jej nemusíme registrovat.

# 3 Separated Presentation

Dnešní klientské, případně RIA aplikace pracují tak, že z nějakého zdroje získají data, ta pak zobrazí v uživatelském rozhraní aplikace a umožní uživateli data nejen prohlížet, ale také modifikovat. Upravená data jsou pak zkontrolována a odeslána zpět ke zdroji pro jejich uložení. Celý tento cyklus ovšem vyžaduje určitou aplikační logiku a definování této logiky na úrovni uživatelského rozhraní jej pak tvoří těžkopádným a složitým na pochopení. Hlavním problémem je však složitost správy, úprav, testování a znovu užití takto vytvořených částí aplikace.

Návrhový vzor *Separeted Presentation* pomáhá k jasnému oddělení vizuální reprezentace dat od aplikační logiky. Takovéto oddělení aplikační logiky (dále jen logiky) od uživatelského rozhraní (dále jen rozhraní) má mnoho výhod. Můžeme například logiku testovat nezávisle na rozhraní. Testování rozhraní totiž znamená manipulaci s ovládacími prvky rozhraní, které pak určitým způsobem reagují. I když to vždy neznamená nutnost skutečného klikání na ovládací prvky a s pomocí vhodných testovacích nástrojů se dá manipulace s prvky rozhraní nasimulovat, je nastavení těchto nástrojů časově náročné. Při testování logiky přes rozhraní je navíc složité oddělit chyby, které byly způsobeny logikou, od těch, které byly způsobeny rozhraním, protože obojí je testováno najednou.

Další výhodou je možnost rozdělit práci na projektu do více týmů. Tým vývojářů se zaměří na realizaci aplikační logiky, zatímco návrháři uživatelského rozhraní se mohou věnovat jeho vývoji nezávisle na aplikační logice. Rozhraní je pak možné snadno upravovat podle měnících se uživatelských požadavků.

Použití vzoru Separated Presentation také pomáhá tvořit aplikace, které jsou snadnější k pochopení a hlavně k pozdější správě. Navíc máme možnost znovu použít kód aplikační logiky na různých místech aplikace. Rozhraní může být například různé pro různé uživatele nebo může být aplikační logika sdílená mezi desktopovou aplikací a RIA verzí aplikace.

## 3.1 Návrhové vzory využívající principu Separated Presentation

Existuje mnoho vzorů, které pracují na principu Separated Presentation. Téměř všechny oddělují uživatelské rozhraní a aplikační logiku do tří oddělených komponent a rozdíly mezi jednotlivými vzory spočívají v tom, jak spolu jednotlivé komponenty spolupracují. Používají se následují typy komponent

- Komponenta pro uživatelské rozhraní - View,
- komponenta pro aplikační logiku - Presenter, Presentation Model,
- komponenta pro byznys logiku a práci s daty - Model.

### Komponenta pro uživatelské rozhraní

Uživatelské rozhraní kompozitní aplikace je většinou složeno z několika View komponent. View chápeme jako skupinu ovládacích prvků, jejichž hlavní funkcí je poskytnout uživateli rozhraní, které odpovídajícím způsobem prezentuje data a logiku práce s daty.

View také předává příkazy uživatele pro manipulaci s daty dalším komponentám. Tedy modelu nebo presenteru, který pak určí, jaký bude mít uživatelův příkaz vliv na uložená data a stav aplikace. V některých případech obsahuje View i část aplikační logiky, která je specifická pro danou implementaci rozhraní, jinak je View definováno jako *UserControl* nebo *DataTemplate* přímo v XAML souboru, bez nutnosti zápisu nějaké logiky do code-behind [3].

### Komponenta pro aplikační logiku

Aplikační logika je postavena nad případy užití Use Case a definuje logické chování a strukturu aplikace, nezávisle na implementaci uživatelského rozhraní. Aplikační logika je často reprezentována komponentou Presenter nebo Presentation Model. Pro umožnění znovu-užití již vytvořené komponenty pro aplikační logiku je nutné, aby tato komponenta neměla přímé reference na třídy uživatelského rozhraní a ovládací prvky a mělo by být možné tuto komponentu nezávisle testovat.

### Komponenta pro byznys logiku a práci s daty

Tato komponenta se stará o získání a správu dat, se kterými aplikace pracuje a zajišťuje také konzistenci a korektnost upravovaných dat. Byznys logika je typicky reprezentována komponentou Model. Model by neměl obsahovat žádnou logiku nebo chování, které je vázáno na určitý scénář Use Case.

### Návrhový vzor Separated Presentation v CAL

Bez ohledu na to, který typ vzoru použijeme, existují dvě možnosti, jak vzor vytvořit. Jedná se o *Presenter-First Composition* a *View-First Composition*.

V případě Presenter-First Composition se jako první vytvoří presenter a ten je pak spojen s vhodným View a modelem. V druhém případě, tedy View-First Composition, se jako první vytvoří View a tomu je pak přiřazen vhodný presenter i model. V obou postupech je využíváno Dependency Injection pro vkládání potřebných závislých komponent.

V aplikacích, které se vytváří pomocí CAL, se využívají varianty nejčastěji používaných vzorů pro Separated Presentation, tedy vzoru Model-View-Presenter. Jsou to Supervising Presenter, Presentation Model a hlavně Model-View-ViewModel. Všechny jsou variantami vzoru Model-View-Presenter,ale navzájem se liší v tom, jak view komunikuje s modelem.

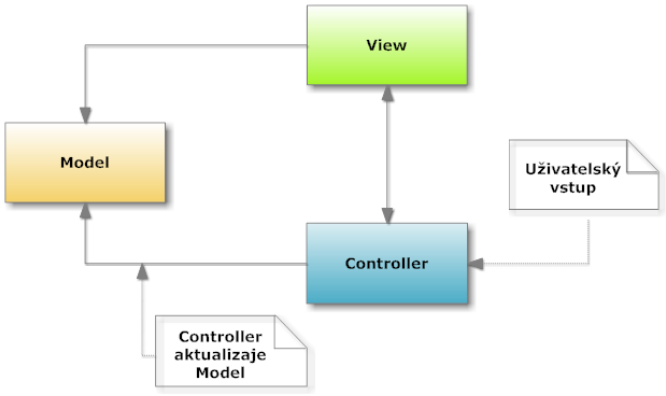
[3] Jedná se o soubor s kódem, který je automaticky vygenerován k odpovídajícímu XAML souboru. XAML soubor má na starosti definici vzhledu dané části aplikace a do code-behind souboru pak zapisujeme kód. Ten obsahuje metody, properties a další, které nějakým způsobem interagují s prvky uživatelského rozhraní. V některých případech se do code-behind píše také kód pro aplikační logiku. Tomu se však snažíme z mnoha důvodů vyhnout.

## 3.2 Historický vývoj

Než ale přistoupíme k popisu jednotlivých vzorů používaných v CAL, je dobré si pro pochopení smyslu těchto vzorů připomenout, jak jednotlivé vzory postupně vznikaly, viz [6].

### 3.2.1 Model-View-Controller (MVC)

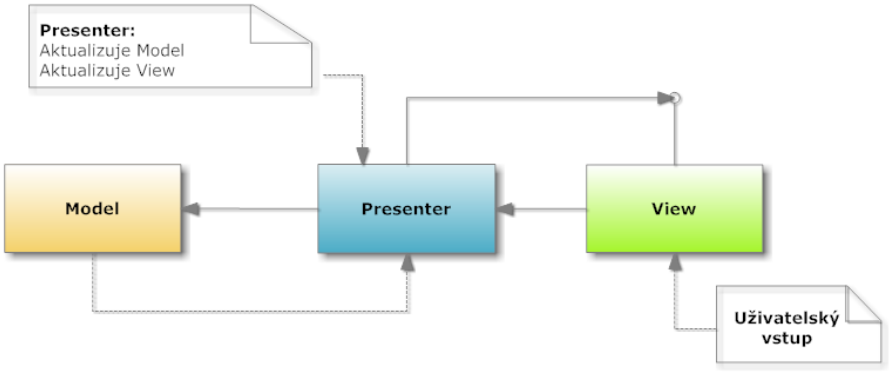
První návrh MVC pochází ze 70. let minulého století, kdy se používaly velké sálové počítače. V té době byly vstup a výstup na sobě do značné míry nezávislé. Pro jednoduchost si můžeme představit, že jeden blok kódu se staral o vykreslení znaků na obrazovku, zatímco jiný zpracovával vstup z klávesnice. Vzor MVC v podstatě odrážel stav hardwaru své doby - View vykreslovalo uživatelské rozhraní a Controller zpracovával vstup (viz Obrázek 5).




Obrázek 5: Model-View-Controller

### 3.2.2 Model-View-Presenter (MVP)

Jak se ale operační systémy a programovací jazyky vyvíjely, vstup a výstup se postupně spojily do jednoho (například tlačítko dnes zvládá jak své vykreslení na obrazovku, tak ošetřování událostí myši, klávesnice a dalších zařízení). Tím v podstatě zanikla původní potřeba pro MVC, protože Controller už nemusel ošetřovat vstup. Také zde se ale postupně přišlo na to, že oddělení aplikačního kódu od definice uživatelského rozhraní je žádoucí, ačkoliv už komponenty teoreticky mohou vstup ošetřovat samy. Právě zde vzniká architektonický vzor MVP (Model-View-Presenter, viz (Obrázek 6).




Obrázek 6: Model-View-Presenter

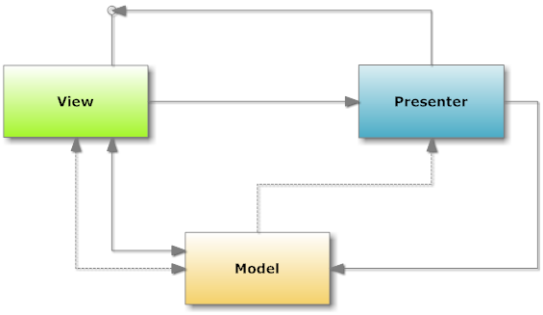
Tento vzor má několik variant:

- Passive View,
- Supervising Controller (SC),
- Presentation Model (PM).

V aplikacích vytvořených pomocí CAL se setkáváme hlavně se SC a PM, proto si je popíšeme.

### 3.2.3 Supervising Controller (SC)

Jedná se již o konkrétní typ návrhového vzoru Separated Presentation, který bývá také označován jako Supervising Presenter, viz [2], a jehož hlavním cílem je oddělit implementaci uživatelského rozhraní od aplikační logiky. Definovat totiž logiku v rámci code-behind uživatelského rozhraní jej činí složitým pro pochopení, správu, testování i případné opakované použití.




Obrázek 7: Supervising Controler

### Použití zvoru SC

Oddělením aplikační logiky od vzhledu uživatelského rozhraní získáme mnoho výhod. Patří mezi ně snadnost testování aplikační logiky, protože odstraníme nutnost klikání na tlačítka a další interakci s ovládacími prvky rozhraní. Bez zbytečných časových a dalších komplikací tak můžeme logiku testovat v izolaci a zaměřit se pouze na chyby v implementaci logiky.

Tím, že je logika od rozhraní oddělena jsem docílili také toho, že můžeme jednou vytvořenou část logiky použít na více místech aplikace. Logika totiž není provázána s konkrétními prvky uživatelského rozhraní, které jsou pouze na jednom místě. Navíc je celá funkčnost popsána na jednom místě a dá se v ní snadno orientovat, provádět úpravy atd. Samostatnost uživatelského rozhraní má samozřejmě také své výhody. Jedná se hlavně o možnost rozdělení práce na aplikaci mezi více týmů. Jeden tým se tak může zabývat pouze implementací aplikační logiky a druhý tým pak bude pracovat na vzhledu uživatelského rozhraní.

Výhody tedy popsány máme. Je ovšem nutné si také ujasnit některá omezení vzoru SC. Tento vzor použijeme v případě, že data, která chceme v rozhraní zobrazovat, podporují databinding a není nutné před jejich zobrazením provádět jejich úpravy a konverze. SC tedy použijeme v případě, že je možné bezpečně provádět změny datech a to s pomocí uživatelského rozhraní a propojení databingem.

### Rešení

Rozdělit zodpovědnost za zobrazování dat v uživatelském rozhraní a koordinaci mezi rozhraním a zdrojovými daty do dvou tříd, a to View a presenter.

*View* má na starosti správu prvků uživatelského rozhraní a chování těchto prvků, které jsou pro rozhraní specifické. View je pomocí data bindingu přímo napojeno na model, který poskytuje přístup k datům a chová se jako pozorovatel tohoto modelu (viz Obrázek 7). View tedy může pomocí bindingu data v modelu měnit a zároveň podle změn v modelu samo sebe aktualizovat. Pokud je však potřeba provádět nějaké komplexnější úpravy modelu, dostává slovo *Presenter*.

Ten dohlíží na interakci mezi View a modelem tak, že provádí úpravy modelu v závislosti na činnosti uživatele a tyto úpravy modelu pak také zobrazuje zpět v uživatelském rozhraní. Presenter proto musí mít referenci na View a často také View má referenci na presenter. Pro účely testování je vhodné, aby mělo view referenci na rozhraní presenteru, ne na jeho konkrétní implementaci. Pak je možné pro testování podstrčit místo úplné implementace presenteru jen její Mock verzi.

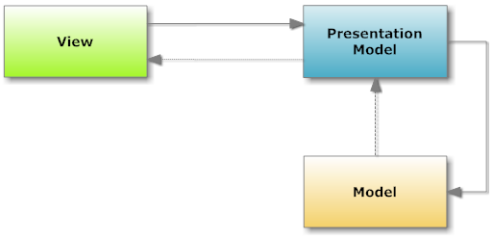
### Nevýhody SC

Všechny nevýhody plynou z onoho oddělení logiky a vzhledu uživatelského rozhraní. Jako každý vzor Separeted Presentation, má i tento několik funkčních prvků. Pro každý prvek to znamená minimálně jeden soubor se zdrojovými kódy. Máme tedy o mnoho více souborů, které musíme spravovat a to může být měné přehledné.

Dalším problémem, který je nutné vyřešit je vytváření View a Presenteru a jejich propojení. Navíc Presenter sice pracuje s modelem, ale model o Presenteru neví. To znamená, že pokud je model změněn jinou komponentou než je presenter, musí se to presenter dozvědět.

### 3.2.4 Presentation Model (PM)

Opět se jedná o konkrétní typ návrhového vzoru Separated Presentation. K problému oddělení uživatelského rozhraní od aplikační logiky přistupuje však jinak než SC.




Obrázek 8: Presentation Model

#### Použití vzoru PM

Vhodnost použití tohoto vzoru je téměř stejná jako v případě SC. Liší se však v práci s daty, která jsou poskytována modelem. SC jsem s výhodou použili v případě, kdy jsem pracovali nad jednoduchými daty. Model pak tato data poskytoval View, které je bez nějakých úprav pouze zobrazovalo.

Oproti tomu PM použijeme v případě, že máme pracovat s daty, která jsou již komplexnější. Jedná se například o data, která musí být před zobrazení ve View nějakým způsobem upravena. Může se tak jednat o přepočet na správné jednotky, či například rozdělení složených dat na data více atomická. Jako příklad může sloužit uchování časového údaje v *MSSQL* serveru. Ten totiž nemá datové typy *DATE* a *TIME*, ale používá datový typ *DATETIME*. Uložený časový údaj je tak kombinací data a času. Tento údaj by tak zřejmě bylo vhodnější rozdělit na samostatné datum a samostatný časový údaj. Přesně toto rozdělení bude mít na starosti PM.

Dále také použijeme PM v případě, kdy chceme provádět validaci dat před tím, než budou uložena do modelu a potažmo do databáze. Může se jednat např. o ověření, že zadaná hodnota náleží do určitého rozpětí nebo zda je daná položka vůbec vyplněna.

#### Řešení

Rozdělit zodpovědnost za zobrazování dat v uživatelském rozhraní a koordinaci mezi rozhraním a daty do dvou tříd, a to view a presentation model.

*View* má na starosti správu prvků uživatelského rozhraní a chování těchto prvků, které jsou pro rozhraní specifické. *PM* má na starosti prezentační logiku a pracuje jako

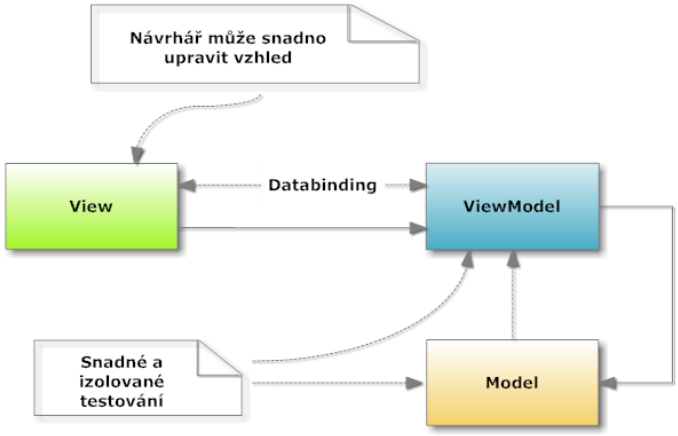
jakýsi prostředník mezi view a modelem (viz Obrázek 8). Udržuje v sobě stav, ve kterém se view nachází, například je-li ve view nějaký seznam prvků, PM pak ví, který prvek je vybrán a reaguje na změny výběru. View se chová jako pozorovatel presentation modelu a ten je zase pozorovatelem modelu samotného. PM by měl poskytovat rozhraní, které pak view může snadno použít, bez nutnosti nějakého složitějšího kódu.

### Nevýhody PM

Ty jsou stejné jako v případě SC.

### 3.2.5 Model-View-ViewModel (MVVM)

S příchodem WPF však vznikl vzor Model-View-ViewModel, který je označován jako varianta MVP. Jedná se o vzor, který je jako stvořený pro moderní aplikace, kde uživatelské rozhraní je prací spíše návrháře, než programátora. Návrhář pomocí deklarativního zápisu v XAML nebo s použitím specifických nástrojů vytváří výsledný vzhled uživatelského rozhraní (dále jen UI) a píše jen minimum klasického kódu. MVVM staví hlavně na WPF mechanismu práce s daty - databindingu.




Obrázek 9: Model-View-ViewModel

#### Použití vzoru MVVM

Vhodnost použití tohoto vzoru je stejná jako v případě PM. Pokud však vytváříme naší aplikaci pomocí WPF či Silverlightu je tento vzor vhodnější než předchozí dva. MVVM byl totiž navržen speciálně pro WPF a View, tedy vzhled uživatelského rozhraní je definován pomocí jazyka XAML a uložen v sepárním souboru. Snažíme se, aby View bylo co nejjednodušší a návrhář mohl snadno měnit jeho vzhled.

MVVM je také vhodné použít v případě, kdy máme několik různých míst, ze kterých čerpáme data. Potom bude VM představovat hlavní VM, který bude propojen pomocí databindingu s View, a ten bude čerpat data z několika podřízených VM. Ty budou obsahovat data z několika zdrojů a hlavní VM bude všechna data agregovat a upravovat do potřebné formy.

#### Řešení

ViewModel (dále jen VM) je, velmi podobný *PresentationModelu* ze vzoru PM. Jedná se tedy o speciální model, ze kterého View čerpá data a který má na starosti veškerou aplikační logiku a manipulaci s daty. Tento model je však v CAL nejvíce používaným modelem, proto je nutné vyzdvihnout jeho hlavní rysy.

VM představuje prostředníka mezi View a Modelem. Jak již bylo řečeno, VM nemusí čerpat data pouze z jednoho datového modelu, ale může jich v sobě zahrnovat více a navzájem jejich data kombinovat do potřebné struktury. Pro získávání dat však VM nemusí využívat pouze datové modely, ale může využívat služeb, které data získavají např. s pomocí nějaké webové služby. Takováto data je však často potřeba upravit, aby mohla být odpovídajícím způsobem dále zpracována a zobrazena ve View. VM tedy provádí potřebné konverze těchto dat, ale také jejich validaci. O validaci se může jednat v obou směrech, jak směrem k View, tak směrem k Modelu.

VM by neměl mít žádný odkaz na View a také by neměl s View přímo komunikovat. Tím bychom totiž vytvořili mezi View a VM silnou vazbu. Té se však snažíme vyhnout z důvodu testování, možnosti použít VM znovu v jiné aplikaci atd. VM by tedy měl obsahovat veškerou funkcionalitu bez nutnosti „sáhnout“ na některý prvek UI. Jsou sice případy, kdy se přímé interakci s prvky uživatelského rozhraní nevyhneme, ale ty jsou spíše výjimkou. Většinou se dá situace vyřešit tak, abychom přímo s ovládacími prvky nemuseli pracovat.

Víme tedy, že VM poskytuje data pro View a to je poté nějakým způsobem zobrazuje. Tato data jsou ve VM definována ve formě *Properties* [4] ke kterým je View připojeno pomocí Databindingu. Binding je vytvořen pomocí deklarativního zápisu v XAML souboru daného View, není k tomu potřeba code-behind. Pro návrháře UI tak je snadné, aniž by musel detailně rozumět kódu, toto propojení mezi View a VM definovat.

Navíc je potřeba propojit View s VM také po funkční stránce. VM má na starosti provádění všech funkcí aplikační logiky a operací s poskytovanými daty. Tyto funkce však musí být nějakým způsobem zavolány a jelikož budou volány na žádost uživatele, musí k tomu docházet za pomocí View a ovládacích prvků UI. Proto jsou jednotlivé funkce, které mají být uživateli přístupné, definovány pomocí tzv. *Commands*. Ty reprezentují propojení mezi ovládacím prvkem UI a funkcí aplikační logiky, která je ve VM implementována. Toto propojení je stejně jako v případě propojení dat řešeno pomocí Databindingu.

Ke zprovoznění Databindingu je nutné splnit ještě jednu podmínku. Tou je nutnost nastavit VM jako tzv. *DataContext* [5] pro dané View. Tímto nastavením tedy umožníme, aby View mohl manipulovat s daty, která VM poskytuje. Může ovšem nastat situace, kdy data nebudou změněna za pomoci View, ale změní je VM sám. Typickým příkladem je získávání dat z webové služby v pravidelných intervalech. View se o tom, že došlo ke změnám v datech VM, které samo View nezpůsobilo, nedozví. Proto je nutné ve VM implementovat rozhraní *INotifyPropertyChanged* [6], které View o změnách v datech informuje. Se změnami v datech souvisí ještě jedna podmínka a tou je skutečnost, že data Modelu může upravovat pouze VM.

[4] Property je veřejně přístupná položka třídy, se kterou lze manipulovat.

[5] DataContext je speciální Property některých ovládacích prvků uživ. rozhraní ve WPF s SilverLightu. Pokud jej danému prvku nastavíme, můžeme pak snadno využívat principu DataBindingu a získávat data pro dané ovládací prvky právě z objektu, který je nastaven jako DataContext.

[6] Rozhraní INotifyPropertyChanged obsahuje událost PropertyChanged, která dává posluchačům na vědomí, která vlastnost VM se právě změnila.

### Použití kódu v code-behind pro konkrétní View

Na internetu je mnoho diskuzí o tomto problému. Hlavní otázkou je, zda lze použít kód v code-behind určitého View nebo je naopak zakázano zde jakýkoliv kód mít. Je nutné si uvědomit, že MVVM je návrhový vzor, který nám dává pouze návod k tomu, jak řešit určitý typ problému. Konkrétně tedy oddělení aplikační logiky od vzhledu UI. Nikde není řečeno, že View nesmí obsahovat žádný kód v code-behind. MVVM nám pouze radí, jak docílit snadno spravovatelného UI z pohledu návrháře, ale také vývojáře. Samozřejmě se v praxi mohou vyskytnout situace, kdy se bez kódu v code-behind neobejdeme. Například ve VM tak definujeme logiku, která rozhodne, za jakých okolností má být určitý prvek View viditelný, a v code-behind daného View pak napíšeme logiku, která skrytí daného prvku zajistí. Umístěním takového kódu do VM bychom vytvořili mezi View a VM silnou vazbu, čemuž se právě aplikací MVVM snažíme zabránit.

### Použití Commands a jejich Parameters

Spouštění metod definovaných ve VM pomocí Commandu, který je k nim přidružen a pomocí DataBindingu je napojen na nějaký ovládací prvek UI, patří k základním rysům vzoru MVVM. V mnoha situacích však budeme potřebovat předat metodě, která je pomocí Commandu volána, nějaký parametr. Je zde více možností, jak toto realizovat.

Představme si situaci, kdy máme v UI zobrazen seznam zákazníků a ke každému z nich je možné si zobrazit detailní informace tím způsobem, že libovolného zákazníka vybereme a klikneme na příslušné tlačítko. Pro předání vybraného zákazníka můžeme použít

1. Element binding, kdy k danému Commandu deklarativně přiřadíme vybraného zákazníka, je CommandParameter (SelectedItem komponenty ListBox) a cílová metoda ve VM bude mít jeden parametr.

2. Ve VM definujeme property SelectedCustomer, která bude pomocí databindingu propojena s vlastností SelectedItem daného ListBoxu. V tomto případě nebude mít cílová metoda parametr žádný, ale interně použije property SelectedCustomer.

Bude výhodnější použít druhou možnost. Použitím element bindingu sice nesvážeme View s VM, ale svážeme jednu vlastnost View s jinou. To však může View učinit obtížnějším pro pozdější spravování. Pokud se později rozhodneme pro jiné rozmístění prvků UI, může dojít k rozbití relativní vazby použité v element bindingu. Pokud použijeme druhou možnost, žádné nebezpečí rozbití relativní vazby nám nehrozí.

# 4 Klíčové komponenty CAL

Jak bylo řečeno v úvodu, CAL nám slouží pro vývoj kompozitních aplikací. Každá aplikace vyvíjená pomocí CAL musí obsahovat určité klíčové komponenty. V následující části si jednotlivé komponenty popíšeme.

## 4.1 Modul

Kompozitní aplikace je složena z několika nezávislých modulů. Každý modul je samostatnou funkční jednotkou, která má na starosti jednu část celé aplikace. Může se například jednat o modul pro logování událostí. Takový modul pak obsahuje samotnou logovací službu, která provádí zápis událostí. Také musí obsahovat třídu, která definuje, jak bude každá událost vypadat, tedy jaké bude mít vlastnosti, metody a chování. Dále pak modul obsahuje View, které je schopno zobrazit jednotlivé zaznamenané události atd. Tento modul je nezávislý na ostatních modulech, ale komunikuje s nimi pomocí slabé vazby.

### Výhody plynoucí z použití modulů

Pokud se rozhodneme pro modulární architekturu vytvářené aplikace, získáme mnoho výhod. To, že jsme funkčnost aplikace rozdělili do modulů, nám umožní izolovaný vývoj těchto modulů. Je pak snadné rozdělit vytváření a správu modulů mezi několik vývojových týmů. Jednotlivé moduly lze testovat v naprosté izolaci od ostatních prvků systému. Tím zajistíme skutečnost, že veškeré chyby nalezené během testování se týkají právě vytvářeného modulu a ne nějaké jiné části aplikace, která je považována za již otestovanou a bezchybnou.

Oddělené moduly je navíc možné snadno verzovat a přidávat do nich nové funkce nebo upravovat funkce stávající. Velkou výhodou v případě použití modulů, je minimalizace času potřebného pro spuštění aplikace. Můžeme totiž vyčlenit určitou skupinu modulů, která je nezbytně nutná pro spuštění aplikace. Tato skupina je pak během spouštění načtena do paměti, ale další moduly jsou načítány až v případě potřeby. Bylo by zbytečné, aby se pokaždé načítaly všechny moduly, když mnoho z nich poskytuje funkce, které uživatel vůbec nemusí použít.

S načítáním modulů se pojí další výhoda. Je totiž možné načítat moduly z různých umístění. Nemusí se tak pokaždé jednat o modul, který byl načten z disku počítače, ale může být získán z databáze nebo z určité webové stránky. To má samozřejmě mnoho výhod, jako je například centralizovaná správa takového modulu. Provedeme-li změnu v modulu, který je načítán z databáze sdílené celou organizací, máme jistotu, že každý uživatel získá odpovídající verzi tohoto modulu.

Navíc je také možné povolit uživatelům přístup pouze k určitým modulům a jejich funkcím. Na základě uživatelských práv a rolí uživatelů tak můžeme selektivně vybírat poskytovanou funkčnost modulů. Uživatel se správcovskými právy tak bude mít přístup k modulům a funkcím, které pro ostatní uživatele nebudou vůbec přístupné nebo budou omezené.

### Pravidla pro tvorbu modulů

Když vytváříme nový modul, měli bychom mít na paměti několik pravidel. Prvním z nich je doporučení, že modul by měl být pro zbytek systému transparentní a inicializovaný pomocí známého rozhraní. Díky tomu zajistíme možnost přidání modulu do nějakého katalogu modulů a můžeme pak se všemi moduly pracovat stejně. Pracovat ve smyslu jejich lokalizace a inicializace.

Dále by pak vyvíjený modul neměl mít přímý odkaz na ostatní moduly ani na aplikaci, která modul načetla. Pokud bychom se totiž v jednom modulu přímo odkazovali na modul jiný, došlo by k vytvoření silné vazby mezi těmito moduly a prakticky veškeré výhody modulárních aplikací bychom ztratili. Proto by mělo komunikace mezi moduly využívat slabé vazby (například pomocí sdílené služby) a ne komunikovat přímo.

Modul by také neměl být zodpovědný za inicializaci všech svých závislostí. Ty by mu měly být dodány externě, například pomocí Dependency Injection. Toto opět slouží pro zajištění slabé vazby mezi jednotlivými moduly a službami, které používají. Jako poslední doporučení uvedeme , že modul by měl být uzpůsoben k tomu, aby se dal do aplikace jednoduše přidat i odebrat.

### Implementace Modulu v CAL

Existuje několik způsobů, jak vytvářet a uchovávat jednotlivé moduly, ale pro CAL je typické vytvořit jednu assembly pro každý modul. Tato assembly pak bude obsahovat všechna view, služby a další třídy potřebné pro daný modul. Také je nutné, aby obsahovala třídu, která implementuje rozhraní *IModule*. Toto rozhraní obsahuje jedinou metodu *Initialize*, která je volání v průběhu inicializace modulu.

```
1  public interface IModule
2  {
3          void  Initialize  () ;
4  }
```

Výpis 4: Zdrojový kód rozhraní IModule

### Životní cyklus modulu

Jak ukazuje Obrázek 10 je životní cyklus modulu rozdělen do následujících částí


Obrázek 10: Životní cyklus modulu

1. Definice/Nalezení modulu - informace o modulu musí být přidány do ModuleCatalogu.

2. Načtení modulu - assembly, která obsahuje daný modul, musí být načtena do paměti. Je možné, že bude nutné tuto assembly nejdříve stáhnout z webu nebo ji jiným způsobem získat.

3. Inicializace modulu - jedná se o vytvoření instance tříd modulu a zavolání Initialize().

### Definice/Nalezení modulu

Informace o jednotlivých modulech, které mohou být v konkrétní aplikaci použity, se nachází v tzv. *ModuleCatalogu*. ModuleCatalog je možné naplnit několika způsoby. První dva způsoby jsou statické a jedná se o *definici modulů v kódu aplikace* nebo *definice modulů v XAML souboru*. Tyto způsoby jsou sice snadné, ale pokud chceme do aplikace přidat další modul, znamená to zásah do jednoho z těchto souborů a rekompilaci celé aplikace.

Proto CAL nabízí další způsoby definice modulů. Jedná se o *čtení informací o modulech z konfiguračního souboru* a *načtení modulů z definovaného adresáře*. Tyto způsoby mají oproti těm předchozím tu výhodu, že je velmi snadné přidat do aplikace nový modul. V prvním případě provedeme požadované úpravy v konfiguračním souboru, který je při každém spuštění aplikace zpracováván *ModuleManagerem*. Musíme však také přidat přidat i vlastní soubor s modulem. Proto se jako nejjednodušší způsob jeví ten poslední a tím je načítání modulů z definovaného adresáře. Pro přidání nového modulu do stávající aplikace pak stačí soubor s modulem, nejčastěji knihovnu dll, zkopírovat do adresáře se stávajícími moduly a restartovat aplikaci.

Ve své ukázkové aplikaci budu používat definici modulů pomocí kódu, proto zde uvádím následující příklad, jak má taková definice vypadat.

```
1  protected override IModuleCatalog GetModuleCatalog()
2  {
3          ModuleCatalog catalog = new ModuleCatalog();
4
5      // Adding modules that are referenced by the shell using typeof():
6      catalog.AddModule(typeof (ModuleA), "ModuleD")
7              .AddModule(typeof (ModuleB))
8              .AddModule(typeof (ModuleC), InitializationMode.OnDemand);
9
10     // Adding modules in code that are not directly referenced by the shell:
11     const string moduleBAssemblyQualifiedName = "Modules.ModuleB.ModuleB, Modules.
           Silverlight, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null";
12
13     catalog.AddModule(new ModuleInfo("ModuleB", moduleBAssemblyQualifiedName));
14
15     return catalog;
16 }
```

Výpis 5: Zdrojový kód metody GetModuleCatalog

### Načtení modulů

Jakmile je naplněn ModuleCatalog, jsou moduly připraveny k načtení a inicializaci. Celý proces načtení a inicializace má na stratosti ModuleManager.

### Inicializace modulů

Inicializace je poslední krok nutný pro přidání modulu do aplikace. Na tomto místě je vhodné provádět následující činnosti. Jednak zde provádíme registrace typů používaných jak modulem samotným, tak i ostatními moduly v containeru. Dále pak přiřazujeme View k jednotlivým regionům, ve kterých se pak budou zobrazovat. To pouze v případě, že se rozhodneme použít techniku ViewDiscovery, viz část 4.4 na straně 39. V poslední řadě pak provedeme integraci modulu do aplikace. Sem například patří přidání ovládacích prvků za pomocí sdílené služby do panelu nástrojů atd.

### Registrace typů v containeru

Během inicializace provádí modul registraci konkrétních implementací obecných rozhraní služeb a tříd, které potřebuje pro svou činnost nebo které naopak nabízí jiným modulům. K tomu, abychom mohli registraci provádět, musí mít modul přístup ke containeru. K tomu použijeme DI, která nám container dodá.

V následujícím výpisu vidíme, že v konstruktoru třídy je uveden container jako závislost a poté jsou v něm pomocí metody *RegisterType()* zaregistrovány konkrétní implementace obecných rozhraní.

```
1  public class TreeModule: IModule
2  {
3          private readonly IUnityContainer container;
4          private readonly IRegionManager manager;
5
6          public TreeModule(IUnityContainer container, IRegionManager manager)
7          {
8              this.container = container;
9              this.manager = manager;
10         }
11
12         public void  Initialize  ()
13         {
14             container.RegisterType<ITreeViewModel, TreeViewModel>();
15             manager.RegisterViewWithRegion(RegionNames.MainRegion, typeof (TreeView));
16         }
17 }
```

Výpis 6: Zdrojový kód třídy TreeModule

### Přiřazení View k regionu

Existují dvě možnosti, jak přiřadit konkrétní View ke konkrétnímu regionu. Jedná se o *View Injection* a *View Discovery*. Rozdíl mezi nimi je popsán v části 4.2.

Ve výpisu 6 vidíme použití metody View Discovery. V tomto případě se při zobrazení regionu MainRegion zobrazí TreeView.

### Integrace modulu do aplikace

Posledním krokem v inicializaci modulu je jeho propojení se zbytkem aplikace. Tento krok se bude samozřejmě lišit pro každou aplikaci, ale je obvyklé například registrovat View daného modulu do navigační struktury aplikace jako jsou menu a panely nástrojů. V tomto kroku se také provádí přiřazení reakcí na událostí vyvolané v jiném místě aplikace a také ke službám využívaným všemi moduly. Například zde tak můžeme reagovat na událost vyvolanou při vypínání aplikace nebo se zaregistrovat k logovací službě atd.

## 4.2 Shell a View

Jedná se o prvky uživatelského rozhraní kompozitní aplikace. Tyto prvky mají na starosti vizualizaci obsahu, který je jim poskytován VM, jenž je těmto prvkům přiřazen.

### Shell

Jedná se o hlavní okno aplikace, ve kterém se zobrazuje nejdůležitější obsah. Shell nejčastěji představuje jedno okno, ve kterém se zobrazuje několik View. Obsahuje pojmenované regiony, do kterých pak jednotlivé moduly mohou vkládat svůj obsah, tedy svá View. Shell navíc obsahuje navigační prvky, jako jsou nástrojové lišty a menu. Rovněž v něm mohou být uvedeny zdroje jako jsou Styles, DataTemplates a další, které se používají nejen v Shellu samotném, ale také v modulech.

V následujícím výpisu vidíme na řádku č. 10 vytvoření pojmenovaného regionu v rámci prvku ContentControl. Jedná se o region, do kterého bude po spuštění aplikace a načtení příslušného modulu vloženo aplikační menu.

```
1  <Windows:Window x:Class="Shell.Views.ShellView"
2        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4        xmlns:prismrgn="clr-namespace:Microsoft.Practices.Composite.Presentation.Regions;
             assembly=Microsoft.Practices.Composite.Presentation"
5        xmlns:Constants="clr-namespace:Infrastructure.Constants;assembly=Infrastructure"
6        xmlns:Windows="clr-namespace:System.Windows;assembly=PresentationFramework">
7         ...
8              <ContentControl HorizontalAlignment="Left" Grid.Row="1" x:Name="MenuRegion"
9              prismrgn:RegionManager.RegionName="{x:Static Constants:RegionNames.
                   MenuRegion}">
10             </ContentControl>
11        ...
12 </Windows:Window>
```

Výpis 7: Příklad definice regionu v Shellu

### View

Představuje složené části UI, které se vkládají do Shellu. View tak představuje kolekci elementů UI. Jedná se však především o jednotku zapouzdřující a oddělující jednotlivé částí UI. Nejčastěji je View reprezentováno klasickým prvkem WPF aplikace, tzv. UserControl (viz Výpis 8). V CAL ovšem View nemusí být definováno pouze pomocí UserControl, ale je možné použít DataTemplates (viz Výpis 9), pomocí nichž bude výsledné View vykresleno na základě dat obsažených v Modelu.

### CompositeView

V CAL je také možné používat tzv. *CompositeView*, kdy se jedná o View, které v sobě obsahuje několik dalších View, označovaných jako ChildView. Nadřazené View se pak má pak na starosti vytvoření všech svých ChildView a výslednou konstrukci CompositeView.

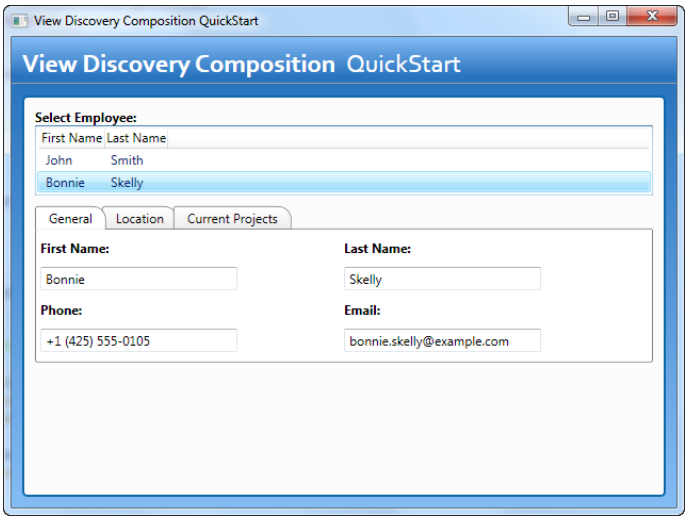
CompositeView bude v praxi vypadat například tak, že se bude jednat o View sloužící pro práci se zákazníky. Toto view pak bude obsahovat dvě ChildView, kdy jedno z nich bude sloužit jako seznam všech zákazníků. Po výběru určitého zákazníka z tohoto seznamu dojde k zobrazení podrobných informací o tomto zákazníkovi ve druhém, detailním View. Jak může takové CompositeView vypadat vidíme na Obrázku 11.

```
1  <UserControl x:Class="WpfApplication1.ExcessivelySimpleView"
2      xmlns="http ://schemas.microsoft.com/winfx/2006/xaml/presentation"
3      xmlns:x="http ://schemas.microsoft.com/winfx/2006/xaml">
4      <Label>A VERY simple view</Label>
5  </UserControl>
```

Výpis 8: Příklad definice View pomocí UserControl

```
1  <Window x:Class="WpfApplication1.Window1" .../>
2      <Window.Resources>
3          <DataTemplate x:Key="ADataTemplateView">
4              <Label>A really simple data template view</Label>
5          </DataTemplate>
6      </Window.Resources>
7      <Grid .../>
8  </Window>
```

Výpis 9: Příklad definice View pomocí DataTemplate




Obrázek 11: Příklad CompositeView

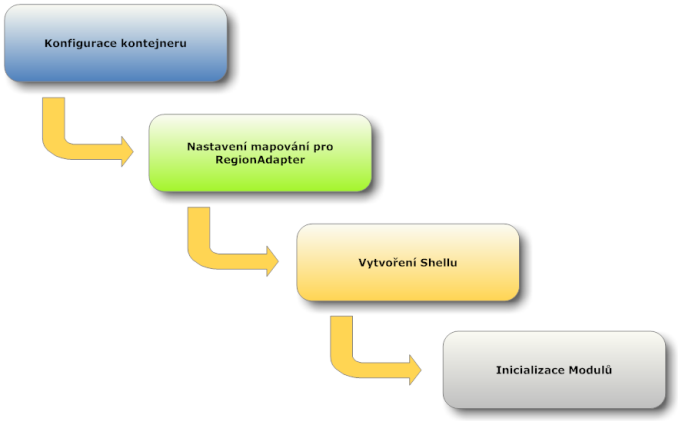
## 4.3 Bootstrapper

Bootstrapper je zodpovědný za inicializaci aplikace vytvořené pomocí CAL. Díky němu máme kontrolu nad tím, jak jsou jednotlivé komponenty CAL napojeny na naši aplikaci.

V klasické WPF aplikaci je v souboru *App.xaml* uveden URI, který je použit pro zobrazení hlavního okna aplikace. V aplikaci vytvořené pomocí CAL je za vytvoření hlavního okna, tedy Shellu, zodpovědný právě Bootstrapper. Je tomu tak proto, že Shell vyžaduje pro svou činnost komponenty jako je například RegionManager. Často také Shell využívá nějakých dalších služeb, specifických pro danou aplikaci, které je nutné mu dodat pomocí DI. CAL má definovanou abstraktní třídu *UnityBootstrapper*, která poskytuje několik virtuálních metod. Tyto metody pak můžeme použít pro vlastní implementaci Bootstrapperu pro naši aplikaci.

### Hlavní fáze Bootstrapperu

Bootsrapper prochází během své inicializace několika fázemi, z nichž ty hlavní jsou uvedeny na Obrázku 12.




Obrázek 12: Hlavní fáze Bootstrapperu

### Konfigurace containeru

Container hraje v aplikacích postavených pomocí CAL i v CAL samotné klíčovou roli. Je nutný pro dodávání všech potřebných závislostní třídám, které je vyžadují. Container je tedy ozančován jako objekt Builder pro DI,viz 2.2 na straně 12.

V metodě ConfigureContainer, jak to ukazuje Výpis 10, probíhá registrace všech důležitých služeb pro naši aplikaci. Bootstrapper zajistí, že nedojde ke dvojí registraci jedné služby, což nám umožňuje upravovat registraci služeb potřebných pro CAL pomocí konfiguračního souboru. Je také možné úlně vypnout registraci implicitních služeb a vybrat si pouze ty, které budeme v naší aplikaci používat.

```
1  protected override void ConfigureContainer()
2  {
3          // Register the implementations for the system interfaces
4          this.Container.RegisterType<IMyWindowService,WindowService>(new
               ContainerControlledLifetimeManager());
5
6          // Register the application model as a singleton.
7          this.Container.RegisterType<IApplicationModel, ApplicationModel>(new
               ContainerControlledLifetimeManager());
8
9          // Register the ViewToRegionBinder
10         this.Container.RegisterType<IViewToRegionBinder, ViewToRegionBinder>();
11
12         // Register the visualizationregistry as a singleton. This has to be a singleton, so all
               registrations are added in this instance.
13         this.Container.RegisterType<IModelVisualizationRegistry, ModelVisualizationRegistry>(
               new ContainerControlledLifetimeManager());
14
15         base.ConfigureContainer();
16 }
```

Výpis 10: Zdrojový kód metody ConfigureContainer

### Nastavení mapování pro RegionAdapters

Během této fáze probíhá mapování implicitních adapterů. Tato mapování jsou pak použita *RegionManagerem* k tomu, aby uměl přiřadit správné adaptéry k regionům definovaným v XAML souboru pro určitá View. Adaptér upravuje chování vizuálních komponent takovým způsobem, aby s nimi mohl RegionManager pracovat. Je samozřejmě možné zaregistrovat také vlastní adaptéry a to pomocí přepsání virtuální metody *ConfigureRegionAdapterMappings()*. Následující výpis ukazuje příklad takového přepsání.

```
1  protected override RegionAdapterMappings ConfigureRegionAdapterMappings()
2  {
3          var regionAdapterMappings = base.ConfigureRegionAdapterMappings();
4
5          // Register an adaptermapping for the NewWindowControl, so this can be used as a
               region.
6          regionAdapterMappings.RegisterMapping(typeof(NewWindowControl),
7          this.Container.Resolve<NewWindowRegionAdapter>());
8
9          //DockingManager Region adapter
10         regionAdapterMappings.RegisterMapping(typeof(DockingManager),
11         this.Container.Resolve<DockingRegionAdapter>());
12
13         return regionAdapterMappings;
14 }
```

Výpis 11: Zdrojový kód metody ConfigureRegionAdapterMappings

### Vytvoření Shellu

V této fázi dojde k zobrazení Shellu. Vytváření Shellu pomocí bootstrapperu usnadňuje testování, protože místo skutečné implementace můžeme použít mock verzi. Ve Výpisu 12 je vidět, že jako VM pro Shell je nastavena nějaká implementace rozhraní *IApplication-Model*. Popis tohoto rozhraní a konkrétní implementace ApplicationModelu je uveden v části 5.1 na straně 47, která se zabývá implementací ukázkové aplikace.

```
1  protected override DependencyObject CreateShell()
2  {
3          // Use the container to create an instance of the shell.
4          ShellView shell = Container.Resolve<ShellView>();
5          var model = Container.Resolve<IApplicationModel>();
6
7          shell.DataContext = model;
8
9          // Display the Shell as the root visual for the application.
10         shell.Show();
11
12         return shell;
13 }
```

Výpis 12: Zdrojový kód metody CreateShell

### Inicializace modulů

Nejdříve je pomocí containeru získán ModuleManager a na něm je vyvolána metoda *Run()*. Tato metoda nejdříve zkontroluje ModuleCatalog a poté na všechny moduly v ModuleCatalogu zavolá metodu *Initialize()*, kterou musí každý modul implementovat (viz 4.1 na straně 30).

## 4.4 RegionManager

RegionManager je komponenta, která je zodpovědná za správu kolekce regionů a za vytváření regionů nových. Nejdříve si však popíšeme, co je to Region a k čemu slouží.

### Region

Region je pojmenovaná lokace v UI, která rezervuje určitý prostor UI a umožňuje v tomto prostoru zobrazovat View jednotlivých modulů. Moduly pak mohou jednotlivé regiony vyhledat a vkládat do nich svá View, aniž by musel vědět, kde se daný region nachází a v jaké formě je zobrazován. To nám snadno umožní změnit rozložení UI bez nutnosti zasahovat do implementace jednotlivých modulů.

Region se vytvoří přiřazením jména učité WPF komponentě. K tomu používáme tzv. attached property jménem *RegionName*, kterou nám poskytuje CAL. Za běhu jsou pak do těchto komponent přidány View nebo skupiny View podle toho, jak je region implementován. Příkladem může být TabControl region, který zobrazuje jednotlivá View jako záložky. K regionům přistupujeme pomocí jejich jména a můžeme tak do nich přidávat View nebo je naopak odebírat. To může probíhat na naši žádost, tedy programově, nebo automaticky. Tyto dva přístupy se nazývají *View Injection* a *View Discovery*.

### View Injection

V tomto případě, v kódu, získáme pomocí RegionManageru referenci na požadovaný region a programově do něj vložíme konkrétní View. Máme tak kontrolu nad tím, kdy jsou View načtena a zobrazena a navíc můžeme View z regionu i odebrat. Není však možné vkládat View do regionu, který ještě nebyl vytvořen. View Injection používáme nejčastěji v případě, kdy chceme reagovat na nějakou uživatelskou akci.

```
1  // View injection
2  IRegion region = regionManager.Regions["MainRegion"];
3
4  var ordersPresentationModel = container.Resolve<IOrdersPresentationModel>();
5  var ordersView = ordersPresentationModel.View;
6
7  region.Add(ordersView, "OrdersView");
8  region.Activate(ordersView);
```

Výpis 13: Zdrojový kód techniky View Injection

### View Discovery

Ve View Discovery nejdříve vytvoříme v RegionViewRegistry spojení mezi názvem regionu a typem View. Když je pak region vytvořen, zjistí, který typ View je k němu přiřazen, a automaticky vytvoří jeho instanci a zobrazí jej. V tomto případě nemáme kontrolu nad tím, kdy je View načteno a zobrazeno.

```
1  // View discovery
2  this.regionManager.RegisterViewWithRegion("MainRegion", typeof(MyView));
3  // View discovery
4  this.regionManager.RegisterViewWithRegion("MainRegion", () => this.container.Resolve<
       ordersPresentationModel>().View);
```

Výpis 14: Zdrojový kód techniky View Discovery

### View Discovery vs View Injection

View discovery je jednodušší přístup k zobrazování View v regionu. Obecně se používá View Discovery. Je však lepší použít View Injection v případech kdy chceme mít nad zobrazováním View z daném regionu větší kontrolu. Díky View Injection tak máme plnou moc nad tím, kdy je View načteno a zobrazeno. Navíc máme také možnost odebrat určité View z regionu, což v případě View Discovery není možné. Pro View Injection se také rozhodněme v případě, že chceme použít více instancí stejného View v regionu. Každé View je pak napojeno na jiná data. Poslední výhodou, kterou View Injection poskytuje je kontrola nad tím, do které instance regionu je View přidáno. Příkladem může být zobrazení View s podrobnými informacemi o zákazníkovi, které chceme přidat do konkrétního regionu pro detaily zákazníka. Těchto reginů pak máme zobrazeno více najednou, například jako záložky prvku TabControl.

### RegionContext

V praxi často potřebujeme do regionu vložit CompositeView (viz paragraf 4.2) a chceme mezi ním a jeho ChildView sdílet nějaká data. Protože každé View je izolováno od těch ostatních, jde o složitou záležitost. CAL nám však poskytuje řešení ve formě RegionContextu. Jedná se o další attached property, kterou CAL poskytuje. Můžeme pomocí něj do regionu uložit libovolnou hodnotu, která se tak stane přístupná všem jeho ChildView, jež jsou v daném regionu zobrazeny. Může se jednat o libovolně jednoduchý i složitý objekt a navíc lze použít i databinding. RegionContext lze použít jak v případě View Discovery, tak i View Injection.

```
1  <TabControl AutomationProperties.AutomationId="DetailsTabControl"
2      cal:RegionManager.RegionName="{x:Static local:RegionNames.TabRegion}"
3      cal:RegionManager.RegionContext="{Binding Path=SelectedEmployee.EmployeeId}"
4      ...>
```

Výpis 15: Přiřazení hodnoty do RegionContextu pomocí XAML

V kódu pak RegionContext nastavíme následujícím způsobem:

```
RegionManager.Regions["Region1"].Context = employeeId;
```

a zpět jej získáme pomocí metody GetObservableContext()

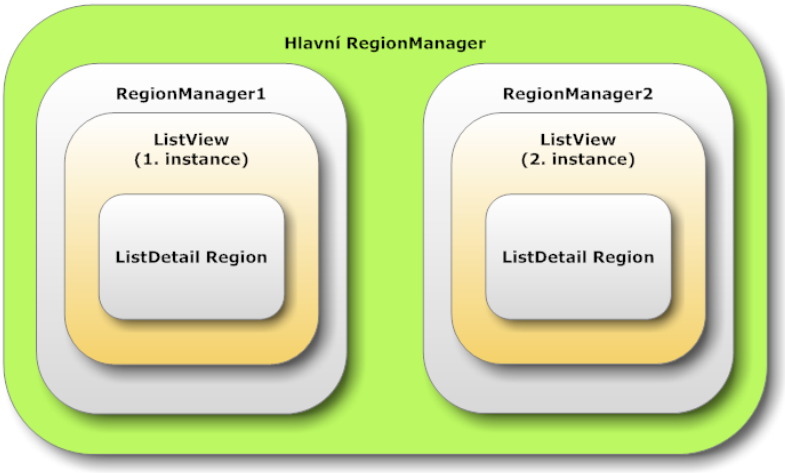
```
this.Model.EmployeeId =
    (int)RegionContext.GetObservableContext(this).Value;
```

RegionContext interně využívá pro uchování hodnoty objektu *ObservableObject*, který implementuje rozhraní *INotifyPropertyChanged*. Je tedy možné reagovat na jeho změny v RegionContextu přiřazením odpovídací reakce na událost *PropertyChanged*.

### Scoped Regions

RegionManager udržuje vazby mezi jménem regionu a typem View. Jména regionu však musí být unikátní a není v tom případě možné mít v RegionManageru registrovám jeden konkrétní region (tedy i jedno konkrétní View) vícekrát. Tato skutečnost se stává problém, pokud například chceme použít TabControl, jenž bude v každé záložce zobrazovat uřčité View, které bude pro všechny záložky stejné, ale bude pracovat s jinými daty.

Musíme proto použít metodu View Injection a navíc tzv. ScopedRegion. V takovém případě dostane každé View svůj vlastní RegionManager a vytvoření vazby na název regionu bude probíhat v tomto unikátním Manageru namísto v tom nadřazeném. Tímto způsobem je tedy možné vytvořit více stejných View zobrazených v jednom TabControlu.




Obrázek 13: Příklad situace využívající ScopedRegionů

Následující výpis uvádí příklad vytvoření ScopedRegionu. Metoda *Add()* vrátí nový RegionManager, který pak View může použít.

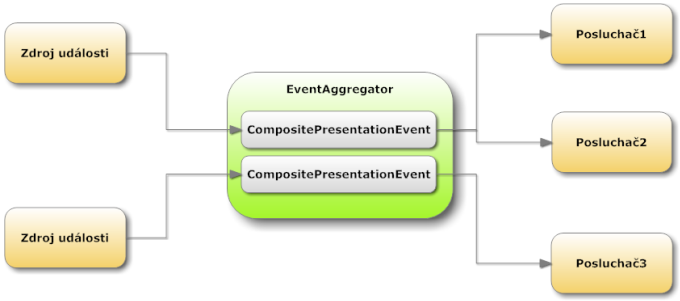
```
1  IRegion detailsRegion = this.regionManager.Regions["DetailsRegion"];
2  View view = new View();
3  bool createRegionManagerScope = true;
4  IRegionManager detailsRegionManager = detailsRegion.Add(view, null,
       createRegionManagerScope);
```

Výpis 16: Vytvoření ScopedRegionu

## 4.5 EventAggregator

Tato komponenta slouží v CAL jako prostředník předávající zprávy (objekty) mezi jinými komponentami, mezi kterými je slabá vazba. Je velmi důležitý pro modulární aplikace, protože nám umožňuje reagovat v jednom modulu na události vyvolané v modulu jiném. Také je velmi výhodný pro budoucí rozšíření aplikace. V takové případtě totiž může být do aplikace přidán další modul reagující na určitou událost a v modulu, který událost vyvolal není potřebovat provádět jakékoliv dodatečné úpravy.

EventAggregator je schopen mít v sobě zaregistrováno mnoho objektů, které vyvolávají události. Samozřejmě také umožňuje více aby vice objektů mohlo naslouchat jedné konkrétní událost. Názorně to ukazuje Obrázek 14.




Obrázek 14: EventAggregator a registrované objekty

### CompositePresentationEvent

Informace, která se přenáší mezi zdrojem a cílem, je pak reprezentována generickou třídou CompositePresentationEvent<TPayload>, která v sobě může nést libovolný generický typ. Díky generice je zajištěno, že jak zdrojový objekt, tak i ten cílový musí používat metody s odpovídajícími parametry. Tedy parametr musí být na obou stranách takový, jaký vstupuje do vytvořené ComopositePresentationEvent.

CompositePresentationEvent obsahuje několik metod. Jedná se o

- Několik variant metody Subscribe,
- Publish,
- Unsubscribe,
- Contains.

Metoda Subscribe má několik přetížení a umožňuje nám tak například pomocí delegáta provádět filtrování událostí a omezit tak případy, kdy bude událost předána dále k cíli. Je také možné nastavit, na jakém vlákně obdrží cíl danou událost. Na výběr máme

- PublisherThread - obdržení na stejném vlákně, které událost vyvolalo.

- BackgroundThread - asynchronní obdržení zprávy na vlákně z Thread-poolu.
- UIThread - obdržení zprávy na vlákně, které má na starost uživatelské rozhraní.

Příkladem konkrétní třídy, reprezentující událost, může být následující kód.

```
public class LogEvent: CompositePresentationEvent<LogEntry>{}
```

Chceme-li vyvolat nebo naopak reagovat na nějakou událost, musíme k tomu použít metodu *GetEvent()*, která je jedinou metodou EventAggregatoru.

Následující výpisy ukazují nejdříve, jak je určitá událost vyvolána pomocí metody *Publish()* a následně na ni v jiném modelu reagujeme pomocí metody *Subscribe*(). Je vidět, že parametr metody Publish je typu LogEntry, jak to vyžaduje událost LogEvent uvedená výše. Stejně tak cílová metoda ShowMessage, přiřazená k metodě Subscribe, očekává parametr typu LogEntry.

```
1  void LogServiceLogChanged(object sender, EventArgs e)
2  {
3          LogEntry last = LogService.GetLastEntry();
4          _eventAggregator.GetEvent<Events.LogEvent>().Publish(last);
5          ...
6  }
```

Výpis 17: Vyvolání události

```
1  public StatusBarViewModel(IEventAggregator aggregator)
2  {
3          _eventAggregator = aggregator;
4          aggregator.GetEvent<Events.LogEvent>().Subscribe(ShowMessage);
5          ...
6  }
7
8  private void ShowMessage(LogEntry entry)
9  {
10         StatusBarMessage = entry.Message;
11 }
```

Výpis 18: Reakce na událost

## 4.6 Commands

Příkaz, tedy Command, nám obecně slouží k vykonávání nějakých akcí vyvolaných uživatelem. WPF nám k tomu nabízí tzv. RoutedCommands. Pro použití v kompozitní aplikaci se však nehodí. Jednak vyžadují, aby vlastní metoda, která má požadovanou reakci na vstup od uživatele vykonat, byla v code-behind daného View. To je v rozporu s doporučeními pro návrhovým vzor MVVM, ale navíc RoutedCommands předávají příkazy pouze v rámci logického stromu [7] a VM, který obsahuje implementaci Commandu, se nachází mimo tento strom.

CAL má ovšem opět řešení, a to *DelegateCommand* a *CompositeCommand*. Oba umožňují předat příkaz mimo hranice logického stromu a nevyžadují reagující metodu v code-behind daného View. Oba implementují rozhraní *ICommand* a dají se tak snad propojit s prvkem UI pomocí Databindingu.

### DelegateCommand

Jméno Delegate získal tento Command díky tomu, že používá delegáty pro vyvolání metody *CanExecute* a *Execute* na cílovém objektu, jakmile dojde k vyvolání Commandu. Jedná se o generickou třídu, takže probíhá ověřování korektnosti parametrů jednotlivých metod, což klasický WPF Command nedělá.

### IActiveAware Interface

V některých případech vyžadujeme, aby se DelegateCommand vykonal pouze, pokud se nachází ve View, které je momentálně aktivní. K tomuto účelu DelegateCommand implementuje rozhraní *IActiveAware*, které má property *IsActive*, jež může být nastavena pokaždé, když se příkaz stane aktivním.

```
1  public class DelegateCommand<T> : ICommand, IActiveAware
2  {
3          public DelegateCommand(Action<T> executeMethod);
4          public DelegateCommand(Action<T> executeMethod, Func<T, bool> canExecuteMethod);
5
6          public bool IsActive { get; set; }
7          public event EventHandler CanExecuteChanged;
8          public event EventHandler IsActiveChanged;
9          public bool CanExecute(T parameter);
10         public void Execute(T parameter);
11         protected virtual void OnCanExecuteChanged();
12         protected virtual void OnIsActiveChanged();
13         public void RaiseCanExecuteChanged();
14 }
```

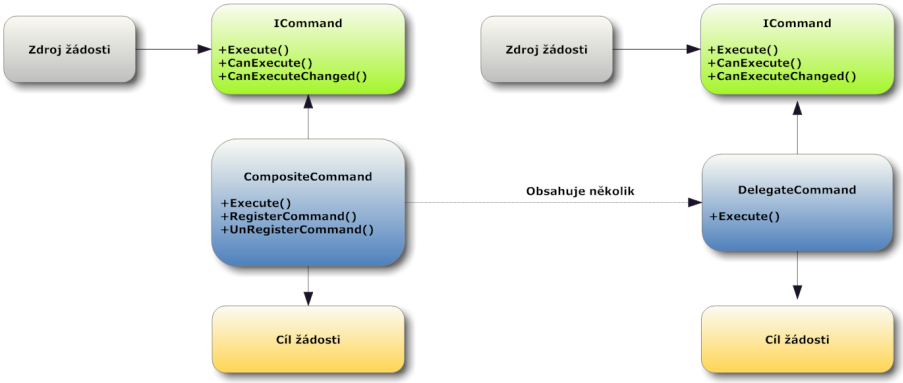
Výpis 19: Definice třídy DelegateCommand

[7] Logický strom - odpovídá zapouzdření elementů zobrazení a jejich datových zdrojů

### CompositeCommand

Jedná se o speciální Command, jenž v sobě zahrnuje několik DelegateCommandů, které se označují jako ChildCommand. Jakmile dojde k zavolání metody *Execute* na CompositeCommandu, tak ten projde celou svou frontu ChlidCommandů a na každém zavolá *Execute*. Stejná situace nastává v případě ověření možnosti vykonat CompositeCommand pomocí metody *CanExecute*.

CompositeCommand tak můžeme například použít v případě, kdy naše aplikace umožňuje mít najednou otevřeno více oken a my chceme dát uživateli možnost, aby tato okna mohl zavřít najednou. Každé okno má samozřejmě možnost být zavřeno samostatně. Pokud však chceme zavřít všechna okna najdou, použije ComopositeCommand. Ten bude složený z jednotlivých DelegateCommand, které mají na starosti uzavření individuálního okna.




Obrázek 15: Znázornění CompositeCommand a DelegateCommand

### Activity Monitoring Behavior

V některých případech budeme chtít, aby se po zavolání metody Execute na CompositeCommand vykonala tato metoda pouze na aktivních DelegateCommandech. K tomu má CompositeCommand implementováno monitorování aktivity jednotlivých ChildCommandů, které uvedeme v činnost, pokud v konstruktoru CompositeCommandu nastavíme parametr *monitorCommandActivity* na hodnotu *true*.

Jakmile je monitorování povoleno, CompositeCommand provede před zavoláním Execute na každý ChildCommand, dodatečné ověření, zda je konkrétní Command aktivní. Pokud aktivní není, property *IsActive* má hodnotu *false*, nedojde k jeho vykonání a nebude ani dotazován pomocí CanExecute zavolaném na rodičovském CompositeCommandu.

# 5 Ukázková aplikace

Ukázková aplikace byla tvořena pro potřeby zadavatelské firmy. Tato firma se v současné době zabývá tvorbou informačních systémů postavených na technologii SmartClient, která je tvořena pomocí .NET frameworku, konkrétně se jedná o technologii WinForms.

Zadavatelská firma má v úmyslu migrovat stávající aplikace do WPF a chce samozřejmě využít kompozitní architekturu a s ní související techniku modulů, jako jí používala doposud. Jejich požadavek tedy byl vytvořit ukázkovou aplikaci, která bude sloužit jako *architektonický vzor* [8] pro migraci jejich stávajících aplikací a také pro tvorbu aplikací nových. Nejde tedy o to, jakou funkčnost má ukázková aplikace, ale o to, jakým způsobem je implementována a jaké navrhové vzory jsou v implementaci použity.

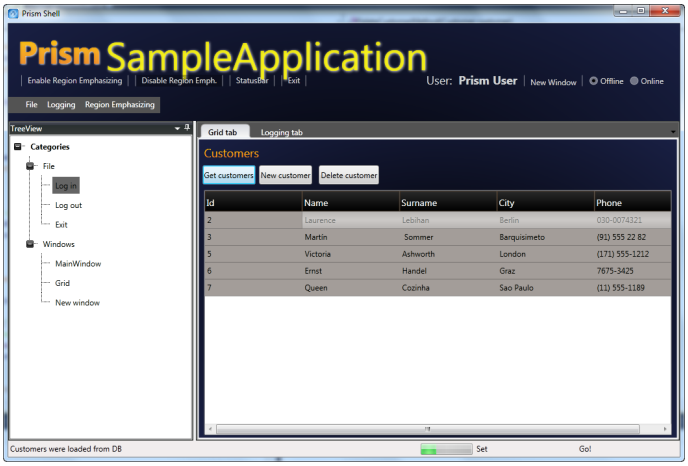
Hlavní požadavky na aplikaci, kterou jsem implementoval, vzešly z aplikace, jakou již firma používá, ale ráda by ji převedla do WPF. Zároveň také zadavatelská firma požadovala, aby aplikace využívala některé komponenty od společnosti Syncfusion, které firma používá ve svých stávajících aplikacích. K tomu, abych tyto komponenty mohl používat jsem samozřejmě potřeboval licenci, která by mi použití umožnila. Proto jsem se pokusil poslat společnosti Syncfusion email, ve kterém jsem jim celou situaci vysvětlil a prosil jsem je o časově omezenou licenci. Společnost se zachovala velmi vstřícně a poskytla mi neomezenou studentskou licenci na jejich balík komponent.

Požadavky byly následující:

- Podpora pro přihlašování a odhlašování uživatelů bez nutnosti restartu aplikace.
- Hlavní okno bude rozděleno do dvou oblastí. Jedna bude obsahovat strom příkazů a druhou oblastí pak bude hlavní pracovní část.
- Aplikace bude umožňovat zobrazení modálních i klasických popup oken.
- Jednotlivá okna reprezentující určitou funkci se budou otevírat do tzv. záložek v hlavní, obsahové, části.
- Použití komponent od společnosti Syncfusion.
- Aplikace bude využívat logovací službu, která může fungovat v režimu aktivním nebo pasivním.

Chtěl jsem také docílit toho, aby se načítaly pouze ty části aplikace, které uživatel momentálně potřebuje. Pokud tedy uživatel aktivuje příkaz otevírající nové View, dojde k vytvoření všech potřebných služeb, VM a dalších komponent až v této chvíli. Tím se sníží doba potřebná pro spuštění aplikace a také náročnost na systémové prostředky, než v případě, kdy se všechny části aplikace inicializují při jejím spuštění.

[8] Architektonický vzor reprezentuje popis řešení strukturalizace softwarového systému tak, aby pracoval co nejefektivněji. Popisuje jednotlivé prvky systému, jejich vzájemnou komunikaci a interakci spolu s omezeními, jak mohou být tyto prvky použity. V podstatě se jedná o návod jak správně rozvrhnout strukturu rozsáhlého systému, jeho rozdělení na jednotlivé podsystémy a funkce, které mají tyto podsystému na starosti atd.




Obrázek 16: Hlavní okno ukázkové aplikace

CAL nám pomáhá vytvářet kompozitní aplikace. Pokud ovšem vytváříme nějakou složitější aplikaci, pak je nám CAL sice dobrý pomocníkem, ale většinou nám samotná nestačí. Klasickým problémem je právě zobrazení View v novém okně aplikace. Ať už se jedná o okno modální, kdy aplikace čeká na jeho zavření, nebo popup okno, kterých může být najednou otevřeno více. CAL samotná nám neposkytuje žádnou možnost, jak takové okno otevřít a hlavně ho pak také zavřít.

Proto jsem se pokusil vyhledat, jestli již někdo podobný problém neřešil, a pokud ano, jaké našel řešení. Narazil jsem na blog jednoho z autorů CAL, Erwina Van Der Valka, který se zabýval řešením hned několika problémů, jež vznikly s požadavky na mou ukázkovou aplikaci.

Pan Van Der Valk na svém blogu zveřejnil dva články, ve kterých se zabýval implementací tzv. Outlook Style Aplikace. Snažil se zde o vytvoření aplikace, která bude v mnohém podobná aplikaci Outlook. Van Der Valk chtěl, aby jeho aplikace splňovala několik požadavků. Některé z nich byly velmi podobné těm, které byly kladeny na mou aplikaci. Mnoho dalších však bylo naprosto opačných. Přesto mi jeho ukázková implementace velmi pomohla, a proto si v následující části popíšeme, jaké jsou základní prvky, které Van Der Valk ve své implementaci vytvořil.

## 5.1 Application model

Jedná se o jakýsi centrální model aplikace, který ví, jak je aplikace strukturována. Jednotlivé moduly mohou tento model používat pomocí rozhraní *IApplicationModel*, které ApplicationModel, dále jen AM, implementuje. AM obsahuje list tzv. Main UseCaseControllers, (viz 5.2), které mouhou být pomocí bindingu napojeny na kolekci tlačítek a poté aktivovány či deaktivovány. AM také obsahuje metodu, kterou je možné libovolný UseCaseController aktivovat z libovolného místa aplikace.

AM používá pro uchovávání kolekce UseCaseControllers tzv. SingleActiveRegion. Ten umožňuje uchování kolekce objektů, které mohou být přidávány, odebírány a akti-

vovány. Jak název napovídá, SingleActiveRegion umožňuje, aby byl v jednom okamžiku aktivní pouze jeden člen jeho kolekce, tedy pouze jeden UseCaseController.

To je ovšem v rozporu s mou ukázkovou aplikací, která umožní mít aktivních několik UseCaseControlles zároveň. Proto bylo nutné SingleActiveRegion nahradit implementací rozhraní IRegion. Ke konkrétní implementaci IRegion se dostaneme v části 5.8. Nyní si vysvětlíme, co onen zmiňovaný pojem UseCaseController znamená.

## 5.2 UseCaseController

Jako *UseCaseControllers* byly v ukázkové implementaci nazvány jednotlivé části této aplikace, které je možné aktivovat a deaktivovat. Každá implementace *UseCaseController*, dále jen UCC, dědí z abstraktní třídy *ActiveAwareUseCaseController*, která definuje potřebné metody a vlastnosti.

Patří sem zejména metoda *OnFirstActivation*, která je vhodným místem pro provedení všech potřebných kroků souvisících s prvotní aktivací UCC. Provádíme zde hlavně přiřazení View k jednotlivým regionům pomocí tzv. *ViewToRegionBinder*. Ten se stará o přidání view do daného regionu a jeho následné odebrání v případě aktivace a deaktivace UCC. Nemusíme proto ručně volat metody region.Add(view) a region.Remove(view). Příklad použití ViewToRegionBinderu vidíme ve Výpisu 20. Zde je ovšem zaražející, proč k regionu přiřazujeme VM a ne samotné View. K vysvětlení se dostaneme v části 5.5.

Hlavní ovšem je, že pomocí ViewToRegionBinderu můžeme jednoduše přiřadit několik View k několika regionům a pak dojde při aktivaci daného UCC k zobrazení všech vybraných view v jejich regionech. Tento přístup má praktické využití v případě, kdy například chceme zobrazit nejen View, které uživateli zobrazí potřebné informace, ale chceme mu také poskytnout nový panel nástrojů umožňující práci s těmito daty.

```
1  protected override void OnFirstActivation()
2  {
3          if (_popUp)
4                  ViewToRegionBinder.Add(RegionNames.PopUpRegion, _loggingViewModel);
5          else
6                  ViewToRegionBinder.Add(RegionNames.MainRegion, _loggingViewModel);
7          }
```

Výpis 20: Použití ViewToRegionBinder pro přiřazení View k regionům

## 5.3 ObjectFactory

Jak jsem uvedl v části 5, chtěl jsem docílit efektivního využití systémových prostředků. K tomu nám pomáhá tzv. *ObjectFactory*, který se stará o vytvoření potřebných komponent pro View až v případě, kdy dojde k jeho první aktivaci. Jak jsme si již všimli, v kompozitní aplikaci vytvořené pomocí CAL velmi často používáme pro dodání potřebných komponent DI, konkrétně *Constructor Injection*. To nám však v tomto případě způsobuje problém, protože veškeré závislosti, které jsou vkládány pomocí Constructor Injection musí být vytvořeny ještě před tím, než bude vytvořena třída, která je používá. My naopak chceme tyto závislosti dodat až v průběhu života třídy, v našem případě UseCase.

Díky ObjectFactory tak můžeme vyjádřit závislost naší třídy na další komponentě v konstruktoru a závislou komponentu vytvořit až v případě potřeby pomocí metody *ObjectFactory.CreateInstance()*. K takto vytvořené instanci se pak dostaneme pomocí property *ObjectFactory.Value*.

Vytváření závislých komponent až v případě potřeby je tedy vyřešeno, ale nelíbí se nám, že bychom měli k těmto závislostem přistupovat pomocí ObjectFactory.Value a ne pomocí konkrétního typu závislosti. Proto má každý UCC metodu *AddInitializationMethods*, ve které můžeme pomocí lambda výrazu přiřadit metodu ObjectFactory.CreateInstance() k členské proměnné třídy. Při první aktivaci UCC pak dojde k uložení instance závislosti do této proměmné a můžeme k ní v kódu přistupovat pomocí typu této závislosti a ne pomocí ObjectFactory.Value.

```
1  public class LoggingMainUseCase : ActiveAwareUseCaseController
2  {
3          private readonly IApplicationModel _applicationModel;
4          private LoggingViewModel _loggingViewModel;
5
6          public LoggingMainUseCase( IViewToRegionBinder viewToRegionBinder,
7          ObjectFactory<LoggingViewModel> factoryViewModel,
8          IApplicationModel applicationModel)
9              : base(viewToRegionBinder)
10         {
11             _applicationModel = applicationModel;
12             AddInitializationMethods(  () => _loggingViewModel = factoryViewModel.CreateInstance())
                   ;
13             AddFinalizationMethods( () => _loggingViewModel = null );
14         }
15          ...
16 }
```

Výpis 21: Použití ObjectFactory pro získání závislosti

## 5.4 Uvolnění použitých prostředků

Také by bylo dobré mít možnost uvolnit použité prostředky, když už UCC nebudeme potřebovat. Například v případě, kdy dojde k odhlášení uživatele. Proto jsem implementaci abstraktní třídy ActiveAwareUseCaseController doplnil o metodu *AddFinalizationMethods*, která umožní programátorům přidat libovolné metody, jež se provedou v případě deaktivace a uzavření UCC. Příklad je uveden ve Výpisu 21.

Navíc bylo nutné do implementace dodat metodu, které umožní uzavření UCC z libovolného místa aplikace. K tomu tedy slouží metoda *Close()*, která nejdříve UCC deaktivuje, pak zruší příznak *initialized* značící, že UCC byl inicializován a odstraní všechny registrace z registru ViewToRegionBinder. To vše zajistí, že při příští aktivaci UCC dojde opětovně k zavolání všech inicializačních metod a správnému použití registru ViewToRegionBinder. Pokud bychom jej totiž nevymazali, došlo by z důvodu dvojí registrace k zobrazení dvou View v daném regionu, což je nežádoucí.

## 5.5 Propojení View s ViewModelem

V jednoduché kompozitní aplikaci, která je vytvořená pomocí CAL je obvyklé propojit View s odpovídajícím VM následujícím způsobem

- V code-behind View zapíšeme v jeho konstruktoru rozhraní VM.
- Při vytváření View využijeme Constructor Injection k dodání konkrétní implementace tohoto VM.
- Získanou implementaci VM nastavíme jako DataContext daného View.

Další možností je použití prostředníka, který vytvoří jak View tak i odpovídající VM a oba je propojí. View i VM jsou pak implementacemi nějakých obecnějších rozhraní, jež jsou při inicializaci modulu, který View obsahuje, nahrazena již konkrétní implementací. VM pak ve svém konstruktoru obsahuje jako parametr rozhraní View jehož modelem bude. Pomocí IoC pak VM vytvoříme, čímž je mu dodána implementace View, a toto View registrujeme ke konkrétnímu regionu.

Je nutné poznamenat, že tato technika se používá spíše u vzoru PM než MVVM. Proto je také v následujícím výpisu použito označení PresentationModel namísto ViewModel.

```
1  public void Initialize ()
2  {
3          RegisterViewsAndServices();
4
5          IStatusBarPresentationModel model = this.container.Resolve<IStatusBarPresentationModel
               >();
6          this.regionManager.Regions[RegionNames.ShellStatusBarRegion].Add(model.View);
7  }
8
9  protected void RegisterViewsAndServices()
10 {
11         this.container.RegisterType<IStatusBarPresentationModel, StatusBarPresentationModel>();
12         this.container.RegisterType<IStatusBarView, StatusBarView>();
13 }
```

Výpis 22: Propojení View s jeho VM (zde s PM)

U ukázkové implementaci však bylo použito jiné řešení. Van der Valk chtěl docílit toho, aby se k regionům nepřiřazovala jednotlivá View, ale naopak ViewModely. Když pak má dojít k zobrazení daného ViewModelu, použije se určité View k jeho vizualizaci. Tento postup si vyžadoval několik komponent.

Jedná se o

- ModelVisualizationRegistry,
- ModelVisualizer,
- Visualizing Region.

### ModelVisualizationRegistry

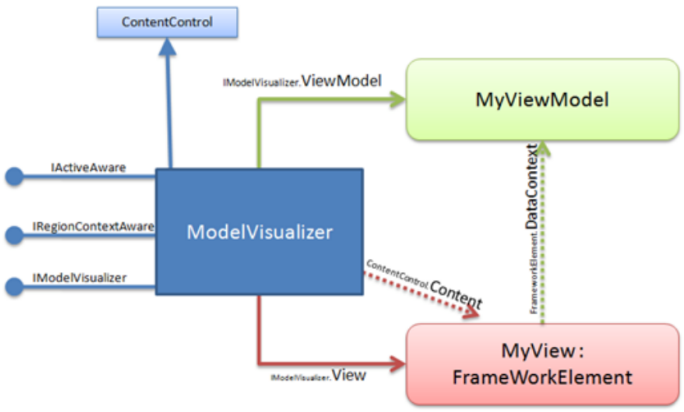
Tato komponenta slouží k přiřazení určitého View k určitému VM. Jakmile pak dojde k zobrazení tohoto VM, bude pro jeho vizualizaci použito View, které je k němu v ModelVisualizationRegistry zaregistrováno. Registraci View a jeho VM provádíme v metodě Initialize pro náš modul.

```
1  public void Initialize ()
2  {
3          // Register the DataService concrete type with the container.
4          _container.RegisterType<IDataService, DataService>();
5
6          visualizationRegistry .Register<GridViewModel, GridView>();
7
8          applicationModel.AddMainUseCase(useCase);
9  }
```

Výpis 23: Zaregistrování View jako vizualizátoru pro VM

### ModelVisualizer - MV

Jedná se o prvek, který je možné vložit do regionu a který je schopen v sobě uchovat jak View tak i jeho VM. MV pak nastaví View jako svůj obsah ve Vizuálním stromu, čímž dojde k zobrazení View v daném regionu. Zároveň je VM nastaven jako DataContext pro View a navíc MV předává mezi View a jeho VM potřebné informace týkající se například RegionContextu nebo hodnot IsActive.




Obrázek 17: Princip objektu ModelVisualizer (převzato z [9])

### Visualizing Region

Visualizing Region je komponenta, která obaluje klasické regiony a umožňuje tak do nich umístit MV. Pokud bychom Visualizing Region neměli, museli bychom v kódu často pra-

covat se samotným MV, zajistit, aby ve správnou chvíli došlo k vizualizaci VM pomocí přiřazeného View atd.

Takto jsou metody MV volány interně právě díky Visualizing Regionu a my můžeme s obsahem regionů pracovat klasickým způsobem.

## 5.6 Zobrazení VM v popup okně

Pokud chceme zobrazit VM v popup okně, je nutné si uvědomit dvě věci.

1. Je možné mít v jednom okamžiku otevřeno několik stejných VM v popup oknech.
2. VM by neměl vědět o tom, že je otevírán v popup okně či v hlavním okně aplikace.

Chceme-li zobrazit VM v popup okně, potřebujeme v tomto okně mít nějaký region, do kterého se vloží odpovídající View pro daný VM. Hlavní problém spočívá v tom, že jména regionů jsou registrována v RegionManageru a každé jméno musí být unikátní. Pokud tedy chceme otevřít několik instancí stejného VM v několika popup oknech, došlo by k pokusu o několikanásobnou registraci stejného jména regionu. To by samozřejmě vedlo k chybě.

Řešením je vytvoření tzv. *ScopedRegionManager*. Toho jednoduše docílíme pomocí příkazu

```
RegionManager.CreateRegionManager()
```

Bylo by opět dobré, aby na tuto skutečnost nemusel programátor myslet. Proto bylo v ukázkové implementaci použito následující řešení. Jelikož se pro tvorbu většiny objektů, které mají nějaké závislé komponenty (jako např. RegionManager), používá IoC kontejner, bylo by výhodné ho použít i pro řešení problému se ScopedRegionMangerem.

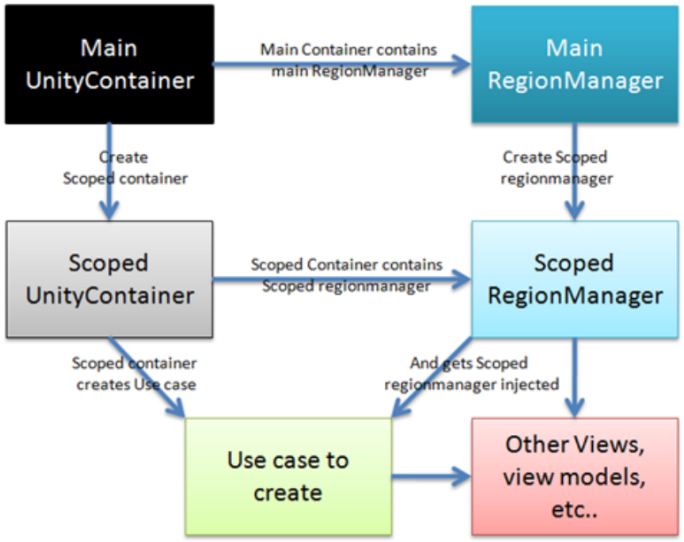
Vytvoříme tedy tzv. scopedContainer, jedná se o nový plně samostatný IoC kontejner, a do něj zaregistrujeme vytvořený ScopedRegionManager. Poté vytvoříme UCC (potažmo VM) v tomto scopedContaineru, a pokud vytvořený UCC bude potřebuje RegionManager, tak automaticky dostane ScopedRegionManager zaregistrovaný ve scopedContaineru.

Jak celá situace vypadá ilustruje Obrázek 18

## 5.7 Zobrazování VM v novém modálním okně

Problém zobrazování VM v novém modálním okně jsem vyřešil implementací služby *MyWindowService*, která je inspirováná následujícím kódem publikovaným na blogu Erwina Van Der Valka. [8]

```
1  public class WindowService : IWindowService
2      {
3          public bool? ShowViewModelInDialog<TViewType>(object viewModel)
4              where TViewType : Control, new()
5          {
6              Window w = new Window();
7              TViewType view = new TViewType();
```




Obrázek 18: Vytvoření scopedContaineru a ScopedRegionManageru (převzato z [9])

```
8                view.DataContext = viewModel;
9                w.Content = view;
10
11               return w.ShowDialog();
12               }
13       }
```

Výpis 24: Zdrojový kód třídy WindowService

Má implementace je doplněna o část, která přizpůsobuje vzhled zobrazovaného okna podle definovaných nastavení. Ta jsou provedena přímo v XAML souboru pro View, jež má na starosti zobrazení dat z daného VM. Jedná se především o nastavení odpovídajících rozměrů okna, jeho výchozí pozice, skrytí tohoto okna v Taskbaru operačního systému atd. Konkrétní implementace služby MyWindowService vychází z následujícího rozhraní

```
1   public interface IMyWindowService
2   {
3           bool? ShowViewModelInDialog(Control view, object viewModel, string windowTitle);
4
5           void CloseView(string viewName);
6
7           Dictionary<string,Control> Views {get;}
8   }
```

Výpis 25: Zdrojový kód rozhraní služby MyWindowService

Vidíme, že toto rozhraní obsahuje dvě metody. První metoda má na starosti ono zobrazení VM v novém modálním okně a ta druhá se pak stará o jeho zavření. Často totiž potřebujeme mít možnost zavřít modální okno jiným způsobem, než jen tím, že uživatel klikne na zavírací tlačítko zobrazeného okna.

Chceme-li nějaké modální okno zavřít, musíme dát službě MyWindowService vědět, o které okno se jedná. Proto si tato služba udržuje seznam zobrazených oken a jakmile jí předáme jméno některého ze zobrazených oken, provede jeho zavření.

Tímto máme tedy vyřešeno jak otevírání klasických popup oken, tak i modálních oken. Dalším požadavkem na ukázkovou aplikaci bylo, aby se jednotlivá okna reprezentující určitou funkci otevírala do záložek v obsahové části hlavního okna. Řešení tohoto požadavku má souvislost také s dalším požadavkem, a tím bylo použití komponent od společnosti *Syncfusion*.

Ve svém balíku komponent pro WPF totiž nabízejí i tzv. *DockingManager*. Jedná se o komponentu, která umožňuje, aby si uživatel přizpůsobil rozmístění jednotlivých prvků aplikace podle svých potřeb. Tento systém je znám např. z Visual Studia od společnosti Microsoft, kde je možné jednotlivé části UI přesouvat, dokovat, měnit jejich velikost atd. Navíc má tato komponenta podporu pro tzv. MDI a TDI, což je přizpůsobení UI pro práci s více okny, panely a záložkami. Použití záložek bylo dalším požadavkem na ukázkovou aplikaci, takže začlenění DockingManageru do implementace působilo velmi slibně. Bylo ovšem nutné přizpůsobit některé části CAL a také vytvořit podporu pro zobrazování UCC v záložkách DockingManageru.

## 5.8 Zobrazení UCC v DockingManageru

Nejprve bylo nutné zajistit, aby DockingManager uměl pracovat s CAL regiony. Proto bylo potřeba vytvořit RegionAdapter, viz 4.3, který by upravoval chování jednotlivých prvků DockingManageru tak, aby mohly spolupracovat s RegionManagerem. Toto však společnost Syncfusion vyřešila sama, protože pro DockingManager napsala DockingRegionAdapter.

Tímto bylo vyřešeno umístění jednotlivých regionů do dokovacích panelů, se kterými se pak dá libovolně manipulovat. Bylo tedy možné vytvořit region pro strom příkazů, který byl jedním z požadavků na ukázkovou aplikaci. Do tohoto regionu je při spuštění aplikace umístěno View, které je napojeno na VM s jednotlivými příkazy. Toto View má ve svém XAML souboru nastaveny všechny vlastnosti potřebné pro umístění do DockingManageru. Je zde například řečeno, že strom bude zobrazen jako tzv. Dock panel, bude na levé straně UI a bude zabírat definovanou šířku. Protože tento strom bude uživatel potřebovat po celou dobu, kdy bude s aplikací pracovat, bylo v XAML souboru také zakázáno zavření tohoto panelu.

Podpora pro funkčnost dokovacích panelů jako CAL regionů byla tedy zajištěna. Bylo ovšem nutné zajistit i podporu pro záložky zobrazované v obsahové části aplikace. Především se jednalo o ten případ, kdy uživatel již má některý UCC v záložce otevřen a celkově je v obsahové části záložek, reprezentujích další UCC, více. Pokud v této chvíli uživatel vybere ze stromu příkazů takový příkaz, který aktivuje již otevřený UCC, bylo by jistě žádoucí, aby se záložka s tímto UCC stala také aktivní a zobrazila se uživateli.

Protože práci se záložkami má na starosti DockingManager a aktivaci View pro vybraný UCC zase RegionManager, bylo nutné zajistit spolupráci mezi těmito objekty. Tento problém jsem vyřešil implementací vlastního Regionu, tzv. *DockingDocumentRegion*, jehož nejdůležitější metodou je metoda *Activate*, jak ukazuje následující výpis.

```
1  public class DockingDocumentRegion: Region
2  {
3      public override void Activate(object view)
4      {
5          IDockableUseCase useCaseController = view as IDockableUseCase;
6          if (useCaseController != null && useCaseController.DockableViewModel != null)
7          {
8              IDockableViewModel viewModel = useCaseController.DockableViewModel;
9              _dockingManager.ActivateWindow(viewModel.Name);
10         }
11         base.Activate(view);
12     }
13 }
```

Výpis 26: Zdrojový kód objektu DockingDocumentRegion

DockingDocumentRegion tak zajistí, aby byl libovolný UCC, konkrétně View vykreslující informace z odpovídajícího VM, zobrazen jako záložka.

Tím však vznikla potřeba nějak definovat, že konkrétní VM poskytuje data, která pak View vykreslí do záložky. Záložka má v UI definován také svůj titulek, podle kterého by měl uživatel poznat, jakou funkčnost záložka reprezentuje. Tato informace by také měla být ve VM obsažena. Navíc DockingManager nemusí nutně zobrazovat vybraný prvek jako záložku, ale může se jednat o libovolný typ dokovacího panelu. Bylo by tedy velkou výhodou definovat ve VM, která z těchto možností bude pro vykreslení View použita.

Pro tyto účely jsem vytvořil rozhraní IDockableViewModel, zajišťující pro VM, který toto rozhraní implementuje, jeho korektní zobrazení v DockingManageru. Pokud nějaký VM toto rozhraní implementuje, dojde během propojování tohoto VM s odpovídajícím View pomocí objektu ModelVisualizer (viz 5.5), k nastavení potřebných vlastností pro DockingManager.

```
1  public interface IDockableViewModel
2  {
3      // header of the Dockwindow or DockDocument tab
4      string Header { get; }
5      // desired state of docked view
6      DockState DockState { get; }
7      // this name is necessary for DockingManager window activation
8      string Name { get; }
9  }
```

Výpis 27: Zdrojový kód rozhraní IDockableViewModel

DockingManager má navíc schopnost uložit si rozložení jednotlivých dokovacích panelů a toto rozložení pak také v libovolné chvíli obnovit. Je k dispozici několik možností, jakým způsobem bude rozložení uchováno. Je tak například možné uložit rozložení do registrů systému, do XML souboru, do souboru binárního atd. Ukázková aplikace si rozložení svých panelů ukládá při svém ukončování a načítá jej při svém startu, ale bylo by možné tento stav upravit tak, aby se rozložení ukládala pro jednotlivé uživatele zvlášť. Pak by si každý uživatel mohl aplikaci přizpůsobit podle svých potřeb a po každém přihlášení by bylo jeho rozložení automaticky načteno.

Nyní jsou již vyřečeny veškeré problémy se zobrazování VM, respektive UCC. Umíme je zobrazit jak v normálním popup okně, tak v okně modálním, které čeká na zavření. Dokážeme VM zobrazit v dokovacích panelech a záložkách a následně již zobrazené záložky aktivovat.

Můžeme tedy přistoupit k problému přihlašování a odhlašování uživatelů. Jedná se sice o jeden z prvních požadavků, který byl na ukázkovou aplikaci kladen, ale v kompozitní WPF aplikaci je k vyřešení tohoto problému potřeba nejprve vyřešit ostatní problémy, které s tímto souvisí.

## 5.9 Přihlašování a odhlašování uživatelů

Uživatel je v systému reprezentován objektem User, který má několik properties, z nichž ta hlavní je isLogged a je typu Boolean. Podle této property pak v AM, viz 5.1, a případných dalších VM, poznáme, že je uživatel přihlášen a můžeme mu tak povolit či zakázat jednotlivé prvky ve stromu příkazů. Jak bylo řečeno v části 4.6 každý DelegateCommand obsahuje metodu *CanExecute*, jejíž výsledek typu boolean říká, zda je daný příkaz vykonatelný.

Chceme-li tedy povolit libovolný příkaz pouze pro přihlášeného uživatele, stačí do těla metody CanExecute daného příkazu dodat podmínku, které ověří, že v AM je vlastnost *isLogged* objektu User nastavena na hodnotu *true*. Krátký příklad nám ilustruje Výpis č. 28, který povolí přihlášení pouze v případě, že uživatel dosud není do systému přihlášen.

```
1  private bool CanLogIn(object notUsed)
2  {
3          return !appModel.LoggedUser.isLogged;
4  }
```

Výpis 28: Zdrojový kód metody CanLogIn

Požadavek byl, aby přihlašování a odhlašování fungovalo *bez nutnosti restartování aplikace*. To znamená, že musíme mít možnost vybrat, které View musíme zavřít, protože byla otevřena přihlášeným uživatelem, a která musí naopak zůstat otevřené, neboť jsou potřebná pro každého uživatele. K tomu nám výborně poslouží AM, který obsahuje seznam všech UCC otevřených v hlavním okně, tedy všech View, která si uživatel otevřel pomocí stromu příkazů. Do původní implementace ovšem bylo nutné doplnit také seznam UCC, která uživatel otevřel ve formě popup oken. Pokud tedy přihlášený uživatel provede odhlášení ze systému, stačí nám projít oba seznamy UCC a na každý UCC zavolat metodu *Close()*.

Tato metoda se postará nejen o zavření všech UCC, které jsou zobrazeny v hlavním obsahovém okně, například ve formě záložek, a všech UCC otevřených v popup oknech, ale provede také všechny tzv. finalizační metody, přiřazené k zavíraným UCC, které mají uvolnit použité prostředky (viz 5.4).

Pro přihlášení uživatele je v AM vyvolána metoda *LogInMethod*, která pomocí služby MyWindowService, viz 5.7, zobrazí přihlašovací okno v novém modálním okně.

```
1  private void LogInMethod(object notUsed)
2  {
```

```
3            Control view = container.Resolve<LogInView>();
4            windowService.Views.Add(ViewNames.LogIn, view);
5            windowService.ShowViewModelInDialog(view, container.Resolve<ILogInViewModel>(), "
                 Log In Window");
6    }
```

Výpis 29: Zdrojový kód metody LogInMethod

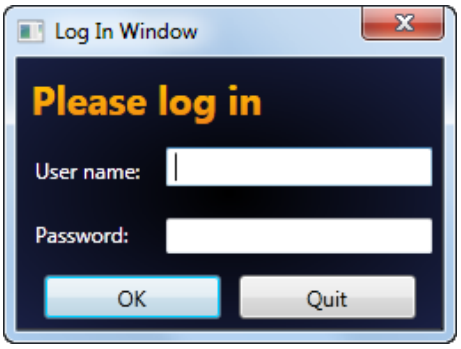
Jak vypadá přihlašovací okno, tedy View, ilustruje Obrázek 19. U tohoto View je zajímavé, že chceme-li docílit toho, aby byl po zobrazení tohoto View umístěn kurzor do textového pole pro vyplnění uživatelského jména, musíme použít code-behind. Situace, kdy potřebujeme umístit kurzor do některého prvku UI, se normálně řeší pomocí objektu *FocusManager* a cílový prvek pro umístění kurzoru zapíšeme do XAML kódu pro dané View.

Takto jsem tuto situaci původně chtěl řešit i já, ovšem z neznámého důvodu FocusManager nastaven pomocí XAML kódu nefungoval správně. Pak jsem se dočetl, že pokud jsou ve View prvky propojeny se zdrojem dat dynamicky pomocí Databindingu, nelze použít FocusManager nastavený pomocí XAMLu[11], ale je nutné použít code-behind a příkaz

```
UserNameBox.Focus();
```

případně

```
FocusManager.SetFocusedElement(this, UserNameBox);
```




Obrázek 19: Okno pro přihlášení uživatele

V ukázkové aplikaci není kladen nějaký důraz na bezpečnost této aplikace, ani nebyly definovány nějaké specifické požadavky pro bezpečnost přihlašování uživatelů atd. Díky modulárnímu systému však nebude žádný problém do aplikace dodat nějakou službu, která zabezpečení zajistí, případně bude definovat pro jednotlivé uživatele jejich role a přístupová práva. Snadno tak bude možné zajistit, že pouze uživatel daného typu bude mít přístup k daným příkazům ve stromu příkazů apod.

Posledním požadavkem na ukázkovou aplikaci byla logovací služba, která bude umožňovat aktivní a pasivní logování. Pojmem pasivní logování je myšleno, že pokud dojde k nějaké

události, která je určitým způsobem významná, zapíše se tato událost do souboru, ale dále se nic neděje. V případě aktivního logování je událost nejen zapsána do souboru, ale dojde také k zobrazení této zprávy v okně aplikace.

Tím, že zapisujeme události do souboru, umožníme všem modulům aplikace aby tento soubor sledovaly a reagovaly na změny v něm způsobené. Bylo by možné také využít EventAggragator, který by všem posluchačům řekl, že došlo k zápisu události do logu a rovnou jim tuto událost předal, ale použití souboru, který je moduly pozorován, mi připadalo jako obecnější řešení.

## 5.10 Logovací služba

Měl jsem zkušenost s logováním pomocí *Enterprise Library* a její části *Logging Application Block*. Problém byl však v tom, že se mi nepodařilo donutit tuto službu, aby každou zalogovanou událost okamžitě zapsala do souboru. Logovací služba z Logging Application Blocku totiž používá jakýsi typ vyrovnávací paměti a k zápisu do souboru dochází v různých intervalech.

Navíc jsem chtěl, aby logovací soubor měl strukturu, se kterou by bylo možné snadno manipulovat, filtrovat data v ní obsažená atd. Proto jsem chtěl, aby se jednalo o XML soubor, se který lze v jazyce C# snadno pracovat např. pomocí LINQ. To byl další problém Logging Application Blocku, protože při zápisu do XML souboru, který je jako jeden z mnoha možných cílů logování, neustále docházelo k naprostému rozbití formátování, které je pro XML velmi důležité. Výsledný dokument tak byl prakticky nepužitelný.

Proto jsem se rozhodl vytvořit vlastní logovací službu, která by tyto problémy vyřešila. Nejprve jsem tedy vytvořil třídu, která reprezentuje vlastní záznam, jenž se bude zapisovat. Tato třída má několik klíčových properties, jak to ukazuje následující výpis. Je samozřejmě možné snadno přidat další informace, které by mohly být pro zalogování podstatné.

```
1  public class LogEntry
2  {
3          public string Message { get; set; }
4          public DateTime TimeStamp { get; set; }
5          public int  Priority  { get; set; }
6          public int  EventId { get; set; }
7          public string Severity  { get; set; }
8          public string  Title  { get; set; }
9          public int  ProcessId { get; set; }
10         public string  ProcessName { get; set; }
11 }
```

Výpis 30: Zdrojový kód třídy LogEntry

Dále bylo nutné zajistit, aby se každá událost ihned při zalogování fyzicky zapsala do XML souboru. K tomu slouží třída *Logger*, která pro vytvoření XML souboru a práci s ním používá třídu *XDocument* a kolekci objektů a metod z *LinqToXML*. Důležité bylo zajistit, aby při vytvoření logovacího souboru došlo také ke vložení správné XML deklarace a kořenového prvku. Při dalším zápisu události do tohoto logovacího souboru se pouze

za poslední prvek v XML stromu připojí další objekt LogEntry, samozřejmě korektně formátovaný.

Umíme tedy zalogovat libovolnou událost a ihned ji také zapíšeme do souboru. Nyní je potřeba zajistit, aby bylo možné tento soubor pozorovat a jakmile dojde k jeho změně, vyvolat odpovídající reakci. Reakcí je myšleno například přečtení celého stromu XML elementů, tedy celého logu, případně přečtení pouze posledního zapsaného prvku.

Pro sledování logovacího souboru jsem vytvořil třídu *FileWatcher*, která interně používá třídu *FileSystemWatcher*, jež je obsažena v .NET frameworku. Této třídě se při jejím vytvoření předá, pomocí parametru vstupujícího do konstruktoru, cesta a název souboru, který má sledovat. Třída pak poskytuje možnost napojit libovolný handler (obslužnou metodu) na událost, která je vyvolána po každém provedení zápisu do logovacího souboru.

Poté již zbývalo jen vytvořit modul, který by měl logování na starosti. V tomto modulu je implementována služba *LoggingService*, která má na starosti načítání celého logovacího souboru a také získání poslední zapsané události v případě zápisu do logu. Tuto služba pak využívá VM, který data z ní získaná zobrazuje ve svém View. VM definuje několik příkazů typu DelegateCommand, pomocí kterých může uživatel např. smazat celý logovací soubor, vynutit zápis testovací události atd.

Zápis do logovacího souboru může provádět libovolný modul, protože statická třída *Logger*, která má zapis na starosti, je obsažena v infrastruktuře systému a je tedy všem modulům přístupná. Naopak čtení z logu je možné pouze díky službě LoggingService, tudíž jej lze použít pouze v modulu, který tuto službu používá.
Vytvořením logovací služby došlo ke splnění všech požadavků na ukázkovou aplikaci. Jelikož jedním z důvodů proč používat návrhové vzory zde uvedené je snadné testování, rozhodl jsem se do ukázkové aplikace také zahrnout několik testů. Na těchto testech jsem chtěl demonstrovat techniky, které se k testování tohoto typu aplikací používají.

# 6 Testování

V části 2 jsem téměř u každého návrhového vzoru uváděl snadné testování jako jednu z hlavních výhod. K testování se vážou následující pojmy *Mock* a *Stub*. Oba pojmy reprezentují prvek použitý v testovacím procesu, který nám usnadňuje testování komponenty, jež má několik závislostí. Závislosti pro nás mohou představovat dva typy problému

1. Nechceme do testu vnášet chyby, které jsou způsobené implementací závislostí, jež testovaná komponenta používá.
2. Komponenta využívá nějakou službu, která je časově náročná pro testování.
3. Závislost představuje službu, jejíž reálné chování je pro testování nevhodné.
4. Máme sice implementovanou komponentu, kterou chceme testovat, ale zatím ještě nemáme naimplementovány závislosti, které testovaná komponenta bude používat.

V prvním případě, pokud takovou komponentu testujeme, jistě nás zajímá především její chování a chyby, které jsou způsobeny v její implementaci a né implementací závislostí. Proto by bylo dobré, abychom mohli dočasně, pouze pro potřeby testování dané komponenty, nahradit skutečnou implementaci jejích závislostí takovým způsobem, aby v nich nedocházelo k chybám. Jedná se tedy například o nahrazení metod, které provádějí nějaký výpočet, takovým kódem, který vrací napevno nastavené hodnoty.

Ve druhém případě se jedná například o situaci, kdy jako závislost chápeme službu, která komunikuje s databází či souborovým systémem. V takovém případě by každé otestování metody, která službu používá, vyžadovalo připojení k databázi, provedení požadované operace a následné odpojení. To by nám značně prodloužilo čas potřebný pro provedení testu a navíc nežádoucím způsobem zatěžovalo systém.

Typickým příkladem třetího případu je využití mailové služby. Testujeme například, že služba odesílá emaily zákazníkům, kteří si koupili nějaké zboží. Určitě necheme, aby skutečně došlo k zasílání testovacích mailů ke skutečným zákazníkům. Můžeme sice použít nějaký vyhrazený email, ale pak se opět dostaneme ke zbytečnému prodloužení testování, stejně jako ve druhém případu.

Poslední, čtvrtý případ, se používá hlavně v takové situaci, kdy k vývoji používáme techniku tzv. TDD, tedy Test-driven Development. V takovém případě dochází k vývoji aplikace odshora dolů[12]. Pokud tuto techniku aplikujeme na MVVM, pak se jedná o to, že nejprve implementujeme VM, který také chceme testovat, a teprve po úspěšném průběhu testů VM implementujeme služby, které používá.

Nyní již k popisů pojmů Mock a Stub. Jak jsem již uvedl, oba pojmy mají souvislost s testováním komponent, které mají nějaké závislosti. Jak Mock, tak i Stub vystupují v testovacím procesu v roli zástupců za skutečnou implementaci závislostí. Jsou však mezi nimi rozdíly, které si nyní popíšeme.

## 6.1 Mock

Jedná se o zástupný objekt, který nahrazuje nějaký objekt reálně naimplementovaný. Tento zástupný objekt je vygenerován pomocí nějakého mocking frameworku (např. v ukázkové aplikaci použitý *MOQ*). Tento Mock objekt je vytvořen jako součást testu a návratové hodnoty jsou do testu pevně zadány. Každý Mock objekt představuje instanci objektu, který implementuje nějaké rozhraní. Proto je v návrhových vzorech IoC a DI tak často používáno rozhraní jako parametr udávající závislost. Toto rozhraní je totiž možné během testování snadno nahradit právě mock implementací.

Mock reprezentuje tzv. testování chování [13], ve kterém se jedná o ověření, zda byla určitá metoda zavolána, kolikrát byla volána, jestli došlo k vyvolání výjimky během volání metody atd.

Použití mock objektů zahrnuje několik fází.

1. Vytvoření instance mock objektu, který implementuje určité rozhraní.
2. Nastavení chování mock objektu.
3. Vyvolání metod, které otestují chování.
4. Ověření korektnosti chování.

Následující výpis ukazuje postupně jednotlivé fáze testování pomocí mock objektů

```
1  public override void EstablishContext()
2  {
3          ...
4          _mockDataService = new Mock<IDataService>();
5          // do not use DB table but empty collection of Customers
6          ObservableCollection<Customers1> customersCollection = new ObservableCollection<
               Customers1>();
7          customersCollection.Add(new Customers1()
8                                          { City = "TestCity", CustomerID = 0, CustomerName = "
                                             TestName",
9                                            CustomerSurname = "TestSurname", Phone = "123455" });
10         _mockDataService.Setup(ds => ds.GetModel()).Returns(customersCollection);
11         ...
12         _gridViewModel = new GridViewModel(..., _mockDataService.Object,...);
13 }
14
15 public override void Because()
16 {
17         //GetCustomers called from GridVeiwModel invokes GetModel method on DataService
18         _gridViewModel.GetCustomers();
19         ...
20 }
21
22 [Fact]
23 public void When_we_call_GetCustomers_from_VM_DataService_should_perform_
24 GetModel_method_and_fill_Customers_collection()
25 {
```

```
26          // atleastonce because GetModel can be called in two cases
27          // 1 - in GetCustomers
28          // 2 - after NewCustomerCommand was invoked
29          _mockDataService.Verify(ds => ds.GetModel(), Times.AtLeastOnce());
30          Assert.True(_gridViewModel.Customers.Count > 0);
31      }
```

Výpis 31: Části zdrojového kódu pro test třídy GridViewModel

Nejvíce nás budou zajímat body 2 a 4. Implementace těchto bodů se bude pro jednotlivé mock frameworky zřejmě lišit. Zde si popíšeme framework, který jsem použil ve své ukázkové aplikaci, tedy MOQ.

Ve výpisu 31 vidíme,že v těle metody *EstablishContext* nejprve vytvoříme nový mock objekt. Poté definujeme, pomocí metody *Setup*, jak se bude tento objekt chovat, když po něm budeme požadovat vykonání metody *GetModel*. Tato metoda v reálné implementaci provede připojení k databázovému stroji, vykoná dotaz a výsledek vrátí jako kolekci zákazníků. Konkrétně jako objekt ObservableCollection, který obsahuje několik objektů typu Customer. Během testování se však chceme vyhnout komunikaci s databázovým strojem z mnoha, především časových, důvodů.

Komunikaci s databází samozřejmě musíme také otestovat, ale v případě, kdy budeme testovat správnost implementace objektu DataService. Zde ovšem testujeme chování objetku GridViewModel, který DataService používá jako službu. Proto musíme mock objektu říci, aby po zavolání metody GetModel vrátil zpět kolekci programově naplněnou testovacími daty.

V těle metody *Because* dojde během průběhu testu k zavolání metody GetCustomers, obsažené v objektu GridViewModel. Tato metoda pomocí služby DataService načítá kolekci zákazníků z databáze, vytváří z nich upravenou kolekci pro VM a nastavuje prvního člena této kolekce jako property VM *SelectedCustomer*. Tato property je ve View napojena na prvek UI, tzv. ListBox, jehož zvýrazněným a označeným prvkem bude právě SelectedCustomer, tedy první zákazník načtený z databáze.

Následuje testovací metoda, označená atributem *Fact*, ve které ověříme, že došlo k zavolání metody GetModel na mock objektu, a to nejméně jednou. Navíc provedeme kontrolu, že kolekce zákazníků ve VM obsahuje nějaké prvky.

Můžeme si všimnout, že název testovací metody je neobvykle dlouhý a složitý. To má však také svůj důvod. Jakmile totiž náš test spustíme, Visual Studio nám zobrazí seznam všech vykonaných testovacích metod. Ke každé metodě pak přiřadí textovou a grafickou reprezetnaci toho, jak dopadla. Pokud tedy máme testovací metody tímto neobvyklým způsobem pojmenované, můžeme ve výsledku testu snadno poznat, zda je daná funkčnost testované části systému implementována správně nebo je nutné této funkčnosti věnovat pozornost.

## 6.2 Stub

V tomto případě se jedná o ruční vytvoření implementace třídy pro potřeby testování. Vytvoříme kostru skutečné implementace jednotlivých metod rozhraní, které testovací

třída implementuje a jako návratové hodnoty použijeme pevně nastavené hodnoty. Stub tedy není nic jiného než třída staticky definována vývojářem [14].

V následujícím výpisu vidíme stub implementaci rozhraní ILogginService, která místo toho, aby přistupovala k logovacímu souboru na disku počítače, používá staticky vytvořenou kolekci záznamů LogEntry.

```
1  public class LoggingServiceStub : ILoggingService
2  {
3          private bool _cleared;
4          public bool LogCleared { get { return _cleared; }  }
5
6          private int  _getLogCalls;
7          public int  GetLogCalls { get { return _getLogCalls; } }
8
9          public ObservableCollection<LogEntry> GetLog()
10         {
11             ++_getLogCalls;
12             ObservableCollection<LogEntry> entries = new ObservableCollection<LogEntry>();
13             entries.Add(new LogEntry()
14             {
15                 EventId = 1,
16                 Message = "Test",
17                  Priority  = 1
18             });
19             return entries;
20         }
21
22         public void ClearLog() { _cleared = true; }
23
24         public LogEntry GetLastEntry()
25         {
26             LogEntry ret = new LogEntry()
27                                 {
28                                     EventId = 1,
29                                 Message = "Test",
30                                  Priority  = 1
31                                 };
32             return ret;
33         }
34 }
```

Výpis 32: Části zdrojového kódu pro stub implementaci rozhraní ILoggingService

## 6.3 Mock vs Stub

Jak vidíme, hlavní rozdíl mezi Mock a Stub je v implementaci. Pokud použijeme Stub, musíme implementaci testovacího objektu vytvořit sami. Znamená to pro nás nutnost naimplementovat všechny metody, které jsou vyžadovány rozhraním, jež testovací objekt implementuje. Navíc budeme většinou muset přidat další pomocné metody, viz metoda *GetLogCalls* ve Výpisu 32. Pomocí těchto metod budeme moci ověřit, že v průběhu testu došlo k zavolání požadovaných metod na testovaném objektu. Naimplementovat

statický stub se nám hlavně vyplatí v případě, kdy by nastavení potřebného chování pro mock objekt bylo velmi složité a časově náročné.

Oproti tomu pokud potřebujeme otestovat pouze, že určitá metoda testovacího objektu byla během testu zavolána, je pro nás jistě výhodnější použití mock objektu. Ten pro nás zajistí, že všechny metody definované v rozhraní budou v testovacím objektu přítomny. Navíc pokud se jedná o funkci, která nemá návratovou hodnotu, nemusíme na mock objektu ani definovat, co bude funkce vracet. Pouze ověříme, že došlo k zavolání této metody, bez ohledu na její implementaci. Na druhou stranu, pokud chceme testovat pomocí Mock objektů, znamená to pro nás použití nějakého Mock Frameworku. Musíme se tedy s tímto frameworkem naučit pracovat a to vyžaduje další čas.

Závěr je tedy takový, že to, jestli použít pro testování mock nebo stub techniku, záleží na konkrétní situaci. Někdy je rychlejší a méně náročné použít Mock framework, jindy zase vytvoření vlastní stub implementace.

# 7 Závěr

Cílem této práce bylo pochopit a naučit se pracovat s knihovnou *CAL*, Composite Application Library, která slouží pro vytváření kompozitních *WPF* aplikací. Tento dokument spolu s ukázkovou aplikací, která je součástí této práce, bude sloužit jako *architektonický vzor* pro vývoj budoucích aplikací zadavatelské firmy. Nyní totiž firma vytváří kompozitní aplikace vytvořené pomocí technologie *WinForms*, ale chystají se migrovat své aplikace do technologie WPF. Jednak budou migrovat aplikace stávající a navíc chtějí své budoucí aplikace vyvíjet přímo pod WPF.

Prvním krokem této práce bylo seznámení se s návrhovými vzory, které jsou v této knihovně využívány. Jednalo se především o vzory *Inversion of Control* a *Dependency Injection*, jejichž hlavním úkolem je zajistit dodání tzv. závislostí pro objekt, který s nimi potřebuje pracovat. V aplikaci napsané pomocí CAL budeme najčastěji používat DI, viz část 2.2, konkrétně její konstruktorový typ, ale také Service Location, viz část 2.3.

Dalšími velmi důležitými vzory jsou ty, které spadají do oblasti nazývané *Separated Presentation*. Hlavním úkolem těchto vzorů je snaha oddělit implementaci aplikační logiky od konkrétního vzhledu uživatelského rozhraní, detailní popis je v části 3. Sem patří především vzor *Model-View-ViewModel*. Ten se vyvinul ze vzoru Model-View-Presenter, které byl zase následníkem vzoru Model-View-Controller. MVVM je tedy nástupcem vzoru MVP, kterému je principiálně velmi podobný, je však uzpůsobem tak, aby využíval nové možnosti technologie WPF. Jedná se hlavně o DataBinding a Commanding, které jsou opět popsány detailněji v části 3.2.5.

Hlavním problémem, především na počátku této práce, bylo, že vzor MVVM zažíval velký rozpuk a teprve se dostával do povědomí vývojářů. Mnoho z nich vědělo, že tento vzor existuje, ale nebyl dostupný žádný konkrétní návod, jak se tento vzor používá, jaké má mít vlastnosti, co patří k jeho výhodám atd. Bylo dostupných několik článků k tomuto vzoru. Prvním z nich byl článek Johna Gossmana, vývojáře, který pracoval na vývoji první verze aplikace Microsoft Blend. Ten prakticky jako první, spolu s Johnem Smithem, vytvořil vzor MVVM a popsal jeho hlavní výhody, viz [3]. John Smith poté zveřejnil konkrétní příklad, ilustrující použití toho vzoru v reálné aplikaci, viz [4].

Jednalo se však pouze o jednoduchou aplikaci a postupně vyšlo najevo, že je potřeba vyřešit ještě mnoho problémů, aby se tento vzor dal použít u skutečných LOB [9] aplikací. K hlavním problémům patřilo například otevírání nových oken uživatelského rozhraní a jejich následné zavírání. Tomuto problému a jeho řešení se věnuji v částech 5.6 a 5.7.

Velká diskuze se také rozpoutala nad tím, zda může být v code-behind pro komponentu View, která má na starosti vyzualizaci dat z VM, použit nějaký kód nebo ne. Jak ovšem uvádí Glenn Block ve svém článku, viz [5], to, jestli bude v code-behind nějaký kód, záleží pouze na vývojáři a hlavně na dané situaci. Někdy se bez code-behind prostě neobejdeme a jiné řešení by bylo příliš komplikované a někdy je zase použití code-behind nevhodné a je potřeba lépe navrhnout vlastní VM.

[9] LOB aplikace patří do skupiny počítačových aplikací, které se používají v podnikových, zásobovacích a dalších procesech. Jedná se většinou o rozsáhlé programy, které mají velké množství funkcí, pracují s databázemi a často také využívají webových služeb.

Jakmile jsem pochopil teoretický základ, tedy návrhové vzory, začal jsem implementovat ukázkovou aplikaci. Postupně jsem však narazil na další problémy, které jsem nebyl schopen, jen s pomocí CAL, vyřešit. Vlastně se nejednalo o problémy ve smyslu chyb, ale šlo o to, že pouze s použitím CAL je implementace LOB aplikace příliš komplikovaná. Pátral jsem tedy, zda se někdo nezabýval tím, jak LOB aplikací naimplementovat s použitím CAL jednodušeji. Narazil jsem na sérii článků od spoluautora CAL Ervina van der Valka viz [9]. S pomocí jeho ukázkové implementace se mi povedlo naimplementovat ukázkovou aplikaci tak, aby splňovala veškeré požadavky, které si zadavatelská firma stanovila.

Velmi mne také zaujala prezentace, kterou měl John Papa na Microsoft PDC 2009, kde prezentoval použití CAL pro vytvoření rozsáhlé LOB aplikace postavené na technologii Silverlight. V této prezentaci John ukázal použití svého tzv. *Presentation Frameworku*, který ovšem v jeho demostrační aplikaci nebyl zdaleka dokončen. John sliboval, viz [10], že implementací dokončí a zveřejní ji na webu MSDN. Doposud se tak ovšem nestalo, ale pevně doufám, že John svůj slib splní a Presentation Framework spolu s jeho dokumentací zveřejní.

Zadavatelská firma také požadovala, aby byly v ukázkové aplikaci použity komponenty od firmy Syncfusion, se kterými má firma zkušenost získanou během vývoje WinForms aplikací. Velkou výhodou bylo, že se mi podařilo získat akademickou licenci na celý balík komponent. Nebyl jsem tak omezen nějakým časovým limitem, což by znamenalo značnou komplikaci. Během implementace ukázkové aplikace, kdy jsem řešil zapojení Syncfusion komponent, mne zarazilo, že dostupná dokumentace je velmi málo podrobná a obsahuje pouze malé množství příkladů. Musel jsem tak častokrát tápat a zkoušet, dokud jsem na správné řešení nepřišel.

Nakonec se mi však všechny problémy podařilo vyřešit a ukázková aplikace je tak kompletní. Kompletní v tom smyslu, že splňuje veškeré vlastnosti, které zadavatelská firma požadovala. Navíc jsem implementaci doplnil také o ukázky jednoduchých testů, které demonstrují jednu z hlavních výhod kompozitních aplikací, jež jsou vyvíjeny pomocí návrhových vzorů uvedených v části 2, a to *snadné testování* jednotlivých částí výsledné aplikace. Ukázková aplikace nebyla vyvíjena pomocí techniky *TDD*, proto jsou uvedené testy spíše ilustrační a pro reálnou aplikaci by bylo potřeba doplnit více komplikovanějších testů, které by prověřily každou část aplikace.

Je samozřejmě možné tuto ukázkovou aplikaci snadno rozšířit a také vylepšit. Přidání dalších modulů není žádným problémem a stejně tak je jednoduché rozšířit moduly stávající. Dobrým příkladem vylepšení stávající aplikace by například byla služba, která bude přihlášeným uživatelům zajišťovat určitá práva a privilegia. Budou tak například moci používat moduly, které ostaním uživatelům nebudou přístupné, využívat funkce stávajících modulů, které jsou pro uživatele s nižšími právy nedostupné atd.

Implementaci ukázkové aplikace jsem se snažil doplnit co nejvíce komentáři a rovněž jsem v tomto dokumentu používal velkém množství výpisů z kódu. Myslím, že jako podklad pro pochopení principů potřebných pro vytváření kompozitních aplikací, vyvíjených pomocí technologie WPF a knihovny CAL, je tato práce plně dostačující.

# 8 Reference


[1] FOWLER, Martin. *Inversion of Control Containers and the Dependency Injection pattern.* MartinFowler.com [online]. 14.1.2004, 23.1.2004 [cit. 2009-11-07]. Dostupné z WWW: <http://www.martinfowler.com/articles/injection.html>.

[2] FOWLER, Martin. *Supervising Controller* MartinFowler.com [online]. 19.6.2006 [cit. 2009-11-07]. Dostupné z WWW: <http://www.martinfowler.com/eaaDev/SupervisingPresenter.html>.

[3] GOSSMAN, John. *Introduction to Model/View/ViewModel pattern for building WPF apps.* MSDN : Tales from the Smart Client [online]. 8.10.2005 [cit. 2010-03-03]. Dostupné z WWW: <http://blogs.msdn.com/johngossman/archive/2005/10/08/478683.aspx>.

[4] SMITH, John. *WPF Apps With The Model-View-ViewModel Design Pattern.* MSDN [online]. únor 2009 [cit. 2010-03-03]. Dostupné z WWW: <http://msdn.microsoft.com/en-us/magazine/dd419663.aspx>.

[5] BLOCK, Glenn. *The spirit of MVVM (ViewModel), it's not a code counting exercise.* My Technobabble [online]. 3.8.2009 [cit. 2010-03-03]. Dostupné z WWW: <http://blogs.msdn.com/gblock/archive/2009/08/03/the-spirit-of-mvvm-viewmodel-it-s-not-a-code-counting-exercise.aspx>.

[6] BOREK, Bernard. *Úvod do architektury MVC* Zdroják [online]. 7.5.2009 [cit. 2009-12-06]. Úvod do architektury MVC. Dostupné z WWW: <http://zdrojak.root.cz/clanky/uvod-do-architektury-mvc/>.

[7] Patterns & Practices *Composite Application Guidance for WPF and Silverlight - October 2009 [online]* Redmond : Microsoft, 30.10.2009 [cit. 2010-03-07]. Dostupné z WWW: <http://www.microsoft.com/downloads/details.aspx?FamilyID=387c7a59-b217-4318-ad1b-cbc2ea453f40&displaylang=en>.

[8] VAN DER VALK, Erwin. *Erwin van der Valk's blog: Practicing patterns : how to apply a viewmodel to new windows,* How to apply a viewmodel to new windows [online]. 2009 [cit. 2010-01-19]. Dostupný z WWW: <http://blogs.msdn.com/erwinvandervalk/archive/2009/09/18/how-to-apply-a-viewmodel-to-new-windows.aspx>. [webová stránka]

[9] VAN DER VALK, Ervin. *How to build an outlook style application with prism.* Erwin van der Valkś blog: Practicing patterns [online]. 18.9.2009 [cit. 2010-03-03]. Dostupné z WWW: <http://blogs.msdn.com/erwinvandervalk/archive/2009/09/18/how-to-build-an-outlook-style-application-part-1.aspx>.

[10] PAPA, John. *Building a Presentation Framework with Prism for Silverlight* JohnPapa.net [online]. 13.7.2009 [cit. 2010-03-03]. Dostupné z WWW: <http://johnpapa.net/silverlight/building-a-presentation-framework-with-prism-for-silverlight>.

[11] JACKSON, Paul. *Titan:Setting the Initial Focus in WPF.* Titan Is your head in the clouds? [online]. 6.3.2008 [cit. 2010-02-23]. Dostupné z WWW: <http://cloudstore.blogspot.com/2008/06/setting-initial-focus-in-wpf.html>.

[12] SMITH, Dylan. *When should you use Mocks vs Stubs?* Architecture / Agile / TDD [online]. 12.8.2006 [cit. 2010-03-01]. Dostupné z WWW: <http://geekswithblogs.net/optikal/archive/2006/08/12/87801.aspx>.

[13] FOWLER, Martin. *Mocks Arenť Stubs.* Martin Fowler.com [online]. 8.7.2004, 2.1.2007 [cit. 2010-03-01]. Dostupné z WWW: <http://martinfowler.com/articles/mocksArentStubs.html>.

[14] LASSALA, Claudio. *Isolating Dependencies in Tests Using Mocks and Stubs.* CODE Magazine. Květen 2009, č. 6, s. 44-53. Dostupný také z WWW: <http://www.code-magazine.com/Article.aspx?quickid=0906061>.

# A Uživatelská příručka

V této části si popíšeme, jaké podmínky je nutné splnit, aby bylo možné spustit ukázkovou aplikaci a procházet její zdrojový kód. Dále se také seznámíme s ovládáním vlastní aplikace.

## A.1 Spuštění aplikace

Pro spuštění ukázkové aplikace je nutné mít nainstalován následující software:

- Visual Studio 2010 Ultimate Beta 2
- Syncfusion Essential Studio 2009 (version 7) - 7.4.0.20

Pro spuštení testů ve Visual Studiu je nutné mít navíc nainstalováno:

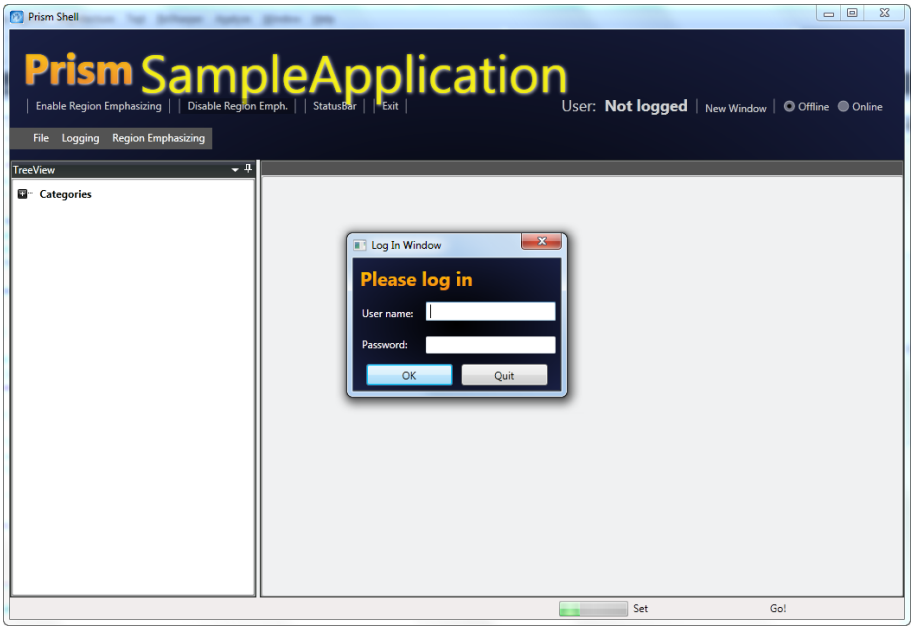
- JetBrains ReSharper 5 Full Edition Pre-Release Build 5.0.1611.9
  http://download.jetbrains.com/resharper/ReSharperSetup.5.0.1611.9.msi
- xunitcontrib 0.3.2g (ReSharper 5 beta + nightly)
  http://xunitcontrib.codeplex.com/releases/view/35006

Dále je nutné v *Configuration Manageru* nastavit *Active Solution Platform* na *x86*. Všechny soubory CAL a další potřebné knihovny jsou obsaženy ve složce LIB.

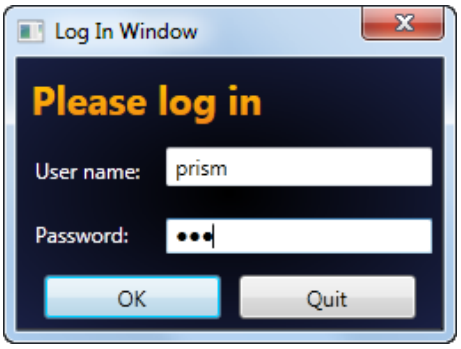
## A.2 Ovládání aplikace

Po spuštění aplikace se zobrazí okno zobrazené na obrázku 20. Bude aktivní okno pro přihlášení uživatele. V něm je nutné do textového pole *User name* zadat hodnotu "prism"a do pole *Password* zadat heslo "123". Poté je možné se kliknutím na "OK"nebo zmáčknutím klávesy "Enter"přihlásit do aplikace. V horní části hlavního okna, pod textem *Prism SampleApplication*, se nachází několik ovládacích prvků. Jsou to tyto prvky:

- Enable Region Emphasizing - kliknutím aktivujeme zvýrazňování regionů. Přejížděním kurzoru myši přes prvky UI můžeme zjistit, v jakém jsou regionu.
- Disable Region Emph. - deaktivace zvýrazňování regionů.
- StatusBar - skrytí/zobrazení statusbaru.
- Exit - ukončení aplikace.
- New Window - otevření popup okna se seznamem zalogovaných událostí.
- Offline - pasivní typ logování.
- Online - aktivní typ logování (zalogované položky budou přidávány do seznamu zalogovaných událostí).




Obrázek 20: Hlavní okno ukázkové aplikace




Obrázek 21: Okno pro přihlášení uživatele

V levé části hlavního okna aplikace se nachází strom příkazů. Strom je možné klasickým způsobem rozbalit a opět sbalit. Jednotlivé položky pak reprezentují určité funkce aplikace a jsou dynamicky aktivovány a deaktivovány podle toho, co právě uživatel dělá a v jakém stavu se aplikace nachází.

- Log in - otevření okna pro přihlášení.
- Log out - odhlášení uživatele (nejdřívě jsou uzavřena všechna otevřená okna).
- Exit - ukončení aplikace.
- MainWindow - zobrazení okna se seznamem zalogovaných událostí.

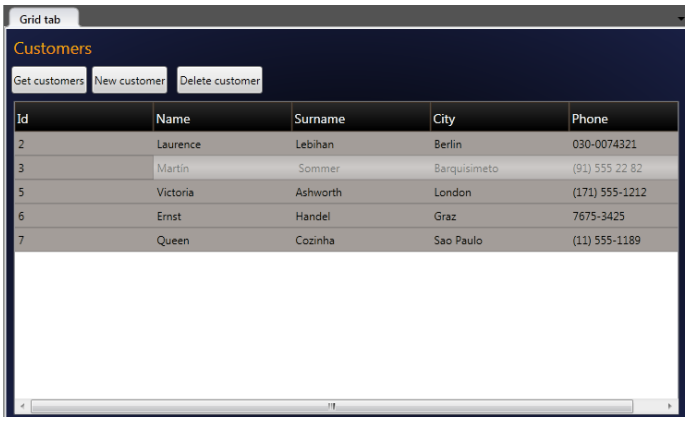
- Grid - zobrazení gridu s daty.
- New window - otevření popup okna se seznamem zalogovaných událostí.

### Práce s oknem pro logování

Toto okno obsahuje dvě tlačítka a seznam zalogovaných událostí. Pokud chceme do seznamu přidat novou událost, je nutné nejprve nastavit typ logování na *"Online"*. K tomu využijeme stejně pojmenovaný ovládací prvek v horní části aplikace. Poté můžeme kliknutím na tlačítko *"Log something!"*vyvolat zalogování uměle vygenerované události a ta bude přidána na konec seznamu. Seznam je také možné vymazat pomocí tlačítka *"Clear Log"*.

### Práce s gridem

Toto okno obsahuje tři tlačítka a grid s daty. Při prvním zobrazení okna je grid prázdný. Data zobrazíme kliknutím na tlačítko *"Get customers"*. Poté můžeme do seznamu přidávat nové položky nebo ty stávající mazat. Smazání položky provedeme tak, že jí nejdříve v seznamu vybereme pomocí kliknutí myší na řádek se záznamem a pak klikneme na tlačítko *"Delete customer"*. Zobrazí se dialog, ve kterém musíme smazání potvrdit kliknutím na *"Yes"*. Tím dojde k odstranění položky ze seznamu i z databáze.




Obrázek 22: Okno s gridem a daty

Přidání nové položky začneme kliknutím na tlačítko *"New customer"*. Zobrazí se nám nové modální okno, ve kterém vyplníme potřebná data a kliknutím na *"Insert customer"*provedeme vložení této položky do databáze. Tlačítko *"Insert customer"*nebude aktivní, dokud každé textové pole s vlastností nové položky nebude obsahovat nějaký text.

# B Obsah přiloženého CD

Struktura CD:

- src - zdrojové kódy ukázkové aplikace
- text - text Diplomové práce
- prirucka.pdf - seznam podmínek pro spuštění a procházení kódem, uživatelská příručka
- readme.txt - popis jednotlivých složek na CD